

高第

Pascal

程式設計技巧

楊洪生 編譯

2211
99

高等 Pascal

程式設計技巧 閱

北京农业工程大学图书馆

年 月 日

楊洪生 編譯



389574

五南圖書出版公司 印行

TP311
94

385574

高等Pascal程式設計技巧

借者 證字 借者 證字

高等 Pascal 程式設計技巧

中華民國 74 年 7 月初版

編譯者 楊 洪 生

發行人 楊 榮 川

發行所 五南圖書出版公司

局版臺業字第 0598 號

臺北市銅山街 1 號

電話：3 9 1 6 5 4 2

郵政劃撥：0 1 0 6 8 9 5-3

印刷所 茂榮印刷事業有限公司

臺北縣三重市重新路五段 632 號

電話：9711628 • 9713227

售價 250 元

(本書如有缺頁或倒裝，本公司負責換新)

簡 介

本書寫作的目的，在於增強業餘或職業程式設計師的 PASCAL 語言方面的知識，並用來解決問題。本書主要強調完整和可用的程式，在建立程式時注意到語言特性、演算法、和資料結構。讀者須先讀過 PASCAL 語言簡介的教科書，也能看得懂 PASCAL 程式表列，並且能寫簡單的 PASCAL 程式。

為什麼本書書名冠上「高等」一詞呢？因為，典型的介紹 PASCAL 的教科書，包括許多小程序，每一個設計指示其所用到的特殊語言元素，其程式都是簡單、無趣的。此外，小程序只是教導語言特性方面的正確使用方式，對於如何利用這些程式特性組成大程式則付之闕如。所以學生僅能把教科書當作工具，並不能從中獲得設計一個真正實用語式的概念。

本書的程式都是真正實用的，每個設計都講求實用性和趣味性，我們也希望這些程式能在讀者的電子計算機上運轉（RUN）。

本書的程式平均比教科書的程式長。較長的程式允許我們將一些較大的任務分成若干小的任務（task），每一個任務實現一相當小而且可了解的程式模組。

有興趣的讀者可任意修飾程式，適用其任意的環境、增加或刪除程式的特性、改進效率或彈性化、等等。事實上，本書的程式非常容易修訂，部分

改進建議附於各章之後，並鼓勵讀者創作及改進。

最後，希望本書可作為技巧智囊。本書的程式反覆的使用成打的模組來解決普通的程式問題。當讀者運轉他們自己寫的這類程式，就可充分利用這些常式而不必再費心思去設計了。

觀察全書

本書最好的部分是寫程式之前，先了解設計決策：為什麼選擇某程式的結構而捨另外一個？在某一固定任務中最佳演算法是什麼？我們如何決定那種方法最好，並能代表現實世界的量、和抽象的資料結構？第一章探討這類設計決策。

本書另外一個主要的理論，是設計一般目的的「工具」常式，這些常式在不同的程式中使用。雖然在寫本書的大部分程式之中發展工具常式，前面若干章集中注意發展工具。第二章設計 CRT 螢幕輸出常式，在後面各章中均使用到；在示範程式 theseus 就利用它在螢幕上建立並解決迷宮的問題。

第三章的常式，是用於交作或由使用人在不同的型態自鍵盤輸入資料，並將某一資料型態轉換成另一種型態。本章的示範程式是 gaslog，它自汽油消費者處接受資料，並提供列印資料的報表。第五章是另一工具內建設計、教導讀者閱讀或設計具彈性而有效率的常式。這些工具都在 print 中使用之（排印本文檔的公用程式，輸出形式是分頁輸出）。

其他各章各自建立一個或二個有趣而且有用的程式：第四章的 calc 由使用人處接收算術公式、並輸出其值。第六章設計 reversi 是一種與電子計算機對抗的遊戲。有二個翻棋（reversi）的版本；其一是在正常本文螢幕上使用，另外一個是利用 APPLE II 的圖形能力。

第七章介紹電子計算機的模擬（simulation），對於了解現實世界系統

而言，是一個强有力的工具。本章內典型代表是 `bouncer`，即球在盒內反跳的樣子：另一程式 `isaac` 是模擬行星在萬有引力下的移動。第八章致力於 `Pascal` 的設計，該程式是深具彈性、有力的「電子工作表報」，對於商界而言甚有價值。

PASCAL 語言

PASCAL 於 60 年代晚期由 Niklaus Wirth 發展出來，他的目的是產生一個適於教學的程式，能夠有很清楚而系統化的概念，並可在大型電腦上使用。很顯然的他獲得相當成就。無論是大型電子計算機、小型電子計算機，甚且個人計算機都相當風行。

PASCAL 為什麼會流行？最主要的原因是 PASCAL 使得工件易於寫、讀，而且修飾程式時遠較許多其他程式語言來的容易。

- PASCAL 有控制結構的敘述（`while repeat for case` 和 `if-then-else`）、程式非常清楚、由上到下的流程設計。如果程式語言沒有控制敘述，則經常要用 `if` 來測試，強迫使用 `goto` 敘述，使得程式難以了解。
- PASCAL 讓設計師把一大型程式打破，分成小的，各自獨立的程序（`procedure`）和函數（`function`），每一個程序或函數都有一固定的任務（`task`）。每一個模組都有其各自的變數，僅在該模組執行時才有用。每一模組與其呼叫常式之間的通信都靠事先定義好的輸入和輸出參數傳遞訊息。這種模組化的目的在使程式易於閱讀而且簡明。
- PASCAL 允許設計師自行定義資料型態（`data type`）和資料結構（`data structure`），充分使用這些特性，使程式更具相容性與易於瞭解。

- PASCAL 允許使用長的變數識別符號 (identifiers)、程序、和函數。(標準 PASCAL 根據前八個字元區分識別符號。) 這使得設計師可用助憶符號命名，也有助於了解程式。
- PASCAL 提供遞迴 (recursive) 程序和函數，換句話說該程序和函數可以自行呼叫本身。這種常式有時為誤用，然而其往往較無遞迴常式的程式易於了解、更簡明。

儘管 PASCAL 是一很好的語言，但不完美。沒有一種廣泛使用的語言是完美的。PASCAL 的檔案能力是最受批評的設計。另外一種說法是，PASCAL 的陣列 (array)、大小 (size) 為其型態 (type) 的一部分，不容易處理任意長的陣列。(新 ISO 標準 PASCAL 已彌補這個問題，但這新版本的 PASCAL 尚未廣泛的有效使用。) 此外，PASCAL 申明一變數是當局變數 (local variable)，當控制權離開該常式時，即被遺忘；所以為了保留該值，必須在呼叫它的常式宣告一全盤 (global) 變數，在必要時可以接達該變數。這些是不完美的缺點，餘容後敘。

無論是好、是壞，程式設計師必須使用一些語言。PASCAL 仍然是一易於發展，而且流行甚廣的語言。

易傳性的特性

一個易傳性的程式，是稍加修改或根本不必改、即能在許多電子計算機上運轉。易傳性的優點很多，後面將費點篇幅討論之。大家都了解，PASCAL 並未明確的定義、教導程式設計師去寫一個可攜帶的程式。

有一些障礙物要易傳，其中有一套包含內建資料型態 (built-in data types) 的限制：最大整數值、實數的精確度和大小，集合內的元素個數等等。每有 PASCAL 的版本都有這些限制，在他們之間就產生了易傳的問題

；例如，為電子計算機寫一個程式，其實數型態假定為8數位，當在另一精度大於8的電子計算機上運轉，該程式即失敗。

其次，許多嚴重的障礙來自語言的實行程式（實行器）。PASCAL 在一特別的電子計算機或作業系統實行，實行器上可以增加一些標準PASCAL上所沒有的特性，使得程式設計師覺得工作更方便、簡單，也可能忽略掉一些比較難實行的特性。這些可能導致標準語言特性，再也說不上標準了。因此，在不同版本之下傳動一個程式，就只能利用語言的共同的特性。如此就產生嚴重的易傳問題，尤其兩個版本相差很大的時候。

第三種障礙來自程式設計師。往往由於電子計算機的能力或限制，使得他們誤入歧途。例如，他們可能用該機器的組合語言寫部分程式，也可能利用他們對電子計算機的記憶體的知识使用變數。他們可能寫一些碼來控制列表機、終端機的特性。

有了這些障礙，程式設計師如何能保證避免易傳性的問題呢？為了使程式達成最大的易傳性，就要注意實行下列技巧：

1. 不管非標準語言的擴充，堅持使用標準版本。
2. 避免使用標準 PASCAL 語言的某些不能於其他機器上使用的特性。
3. 關於電子計算機系統，只有在最基本的前提之下，程式可以運轉（RUN）。
4. 使用「最普通的名稱」，例如，實數的精確度，集合內最大的元素個數，劃底線的字元集之內容與先後次序等等。

這是吸引人的，許多介紹語言的教本以廣大的篇幅討論這些。對於大部分的程式而言，儘可能使其具易傳性是不切實際的。大多數的程式設計師也都發現有使用非標準 PASCAL 語言的必要。就如前面所提的，標準PASCAL 語言擁有一些不當的概念；許多擴充版本就是用來解決這些問題的。如果程式設計師拒絕使用擴充版本，堅持走易傳性的路子，則不但浪費時間

，而且浪費記憶空間，其效率比原始版本還低得多呢！

同理，真正執行時間的限制、記憶體的使用，和程式的審美觀念來看，經常迫使程式設計師寫不可攜帶的程式。

APPLE PASCAL與標準PASCAL

本書的程式使用 APPLE PASCAL 1.1 版的系統，書中提到的 APPLE PASCAL 都是這個版本，因此它可以在 APPLE II PLUS 和 APPLE IIe 上運轉。大部分的程式設計師要運轉 APPLE PASCAL 系統有一個以上的磁碟機，與正常的 40 行螢幕顯示。然而 pascalc 建議使用 80 一行介面卡。列表機則由 print 和 pascalc 控制產生輸出排印。

至於其他版本的 PASCAL 如何使用呢？首先考慮 UCSD PASCAL。APPLE PASCAL 直接由 UCSD II.1 發展出來，二者相似，UCSD II.1 的用戶運轉這些程式只需稍加修改，或者不要修改，最多是改那些針對 APPLE 的特殊硬體特性方面的常式。稍後會警告中止使用 UCSD PASCAL IV.0 版，雖然它增加了某些特性可以改進程式執行狀況。除非特別的申明，本書此後提到的 UCSD PASCAL 指其所有的版本。

APPLE III 的 APPLE PASCAL 與 APPLE II 的 PASCAL 相似，大部分的程式不需改即能運轉，只改那些 APPLE III PASCAL 提供改進的特性部分。

使用其版本 PASCAL 的人，可能必須作大幅度的修改，雖然需要詳細的精確翻譯的指令集，却不是本書討論的範疇；修訂的程度端賴其 PASCAL 特性而定；如果特性相同或相似，則翻譯起來就簡便得多，實際上，許多 PASCAL 的實行，若與標準 PASCAL 的擴充版不同，也是非常相近似。

APPLE PASCAL 和標準 PASCAL 有若干不同之處，主要的是：

字串 (STRINGS) APPLE 和 UCSD PASCAL 除了標準 PASCAL 內的內建資料型態 (built-in data types) 之外，尚提供一內建字串 (string) 資料型態。所謂字串是任意個數字元的，其最大長度是固定的，通常最大長度定為 80 字元，但是最長可達 255 個字元。字串在記憶內儲存方式是一個字元占用一位元組的陣列；第一個位元組 (編號 0) 包含該字串的長度。此外，尚有若干操作字串的功能。字串在其他的版本，也許有意義，但是實行的細節可能不同。

長整數 (LONG INTEGER) APPLE 和 UCSD 除了標準 PASCAL 的正常整數資料型態之外，還提供一種長整數資料型態。APPLE PASCAL 的整數範圍是 - 32767 到 32767，長整數可擁有 36 個位數 (digits)。例如，宣告

< type >

longint = integer [10] ;

longint 型態的變數，其值可能是介於 - 9999999999 到 9999999999 的整數值。長整數是因應程式設計師的需要而定的。許多版本都提供一些廣域的精度型態。

輸入和輸出 (INPUT AND OUTPUT) APPLE 和 UCSD PASCAL 在輸入和輸出領域較標準 PASCAL 更寬廣。它們有為讀或寫磁碟檔的內建常式 (routine)。並有捕捉輸入 / 出錯誤的方法 (例如，硬體失靈，成要讀一個根本不存在的檔案)。程式可能隨機接連檔案，可以讀或寫任意一個結論，不受干涉。有低階 (low-level) I/O 常式直接接達週邊裝置。其他版本也有類似功能的常式。

動態變數 (DYNAMIC VARIABLES) APPLE PASCAL 並無標準 PASCAL 內的 dispose 功能，却有 mark 和 release 內建程序 (procedure

); 它們需要一些不同的管理方法。APPLE 和 UCSD 都提供 `memavail` 函數，它回轉動態變數的有效記憶空間。

圖形 (GRAPHICS) APPLE PASCAL 利用 APPLE 的繪圖能力。

分隔編譯 (SEPARATE COMPILATION) APPLE 和 UCSD 允許一群副常式當作單元編譯。而這些單元可直接讓其他程式使用，不必重新編譯。

段程序 (SEGMENT PROCEDURE) APPLE 和 UCSD PASCAL 允許程式內一些程序宣告成段 (`segment`) 程序。一段程序在程式運轉時不一定要在記憶中；當該程序被呼叫時，即自磁碟讀到記憶裡。程序執行完畢，即被遺忘，其在記憶內所占的空間可由程式的其他部分使用。如此有效的應用記憶體，在小電子計算機使用尤具價值。

雜項截取 (MISCELLANEOUS) APPLE 和 UCSD PASCAL 提供下列常式，使程序設計師的工作簡便：

`moveleft` 和 `moveright` 程序，允許自記憶的一部分抄若干位元組，到記憶體的另一地方。

`fillchar` 程序被填滿一陣列的字元 (或記憶體的任意範圍) 。

`sizeof` 回轉一變數或型態大小的函數。

`exit` 自一程式模組或程式本身依序出口的一程序。

`gotoxy` 是一程序，它將 CRT 螢幕的游標，輸送到一特定的位置。

APPLE PASCAL 同時提供一群額外常式，這些常式常用到的有：

`keypress` 布耳函數，當按下電子計算機的一個鍵，該函數即回轉 **TRUE**，否則回轉 **FALSE**。

`random` 函數，回轉一介於 0 到 32767 之間的虛擬亂數。

`randomize` 程序，將虛擬亂數產生器定初值。

讀者很容易了解，有這些特性的程式，很難在不具備這些特性的版本上運轉。我們並未忽視這問題；為了使程式儘量完美，必要時得犧牲一些攜帶性。茲觀察下列規則以降低易傳性問題到最小。

- 在少數程式模組內的硬體相關有相當的獨立性。例如，只有二個程序被用來控制APPLE的CRT螢幕（其中之一是`gotoxy` 即為APPLE PASCAL和UCSD PASCAL中的內建函數）；將程式移動到另外的電子計算機或終端機，程式設計師只需改本代碼。
- 當一攜帶性解法良好，就不必須用APPLE PASCAL 的非標準特性。（例如，不使用非易傳的`exit`常式，自一程序離開，除非該結果使得代碼簡明或有效率。）
- 在APPLE中的6502組合語言寫的常式，在PASCAL中也使用之，將之翻譯成其他處理機（processor）也較容易。

附錄A 提供解決攜帶問題的詳細建議，但對於少量調整成本書的代碼並不固定。

推薦閱讀（Recommended Reading）

K. Jensen 和 N. Wirth 著的 Pascal User Manual and Report 對於 PASCAL 有全面介紹。本書對於已熟悉電子計算機語言，也作簡單介紹。新的 ISO 標準 PASCAL 設計的非常清晰，有關 ISO PASCAL 的書有：D. Cooper 著的 Standard Pascal User Reference Manual。當我們參考標準 PASCAL 時，上述兩書不可或缺。大部分情況而言，其間並無差別。如有不同之處，只要我們使用到，就會作一交代。

好的 PASCAL 教科書有：Programming in Pascal（P. Grogono

著) ; Introduction to Pascal (J. Welsh and Elder 著) 。

除了介紹階層的書籍之外，尚有二本傑作：一是N. Wirth 著的 Algorithms + Data Structure = Programs，另一是B. Kernighan 和 P. J. Plauger 合著的 Software Tool in Pascal 。

高等PASCAL 程式設計技巧

目 次

1	什麼是好程式	1
	使用者的準則.....	2
	程式設計師的準則.....	7
	程式語言.....	10
	推薦閱讀.....	11
2	陰極射線管技術	13
	theseus 常式.....	22
	建立迷宮.....	25
	解答迷宮.....	31
	建 議.....	35
	推薦閱讀.....	36
3	交作輸入	37
	標準的 PASCAL 本文輸入	37
	鍵盤輸入.....	38

字串輸入.....	41
字串操作常式.....	46
gaslog 常式	47
資料結構.....	54
字串序連.....	56
布耳輸入.....	56
定點數值的輸入.....	58
接達字串字元.....	63
日期輸入.....	64
資料檔案初域.....	66
資料登錄.....	70
產生報表.....	73
建 議.....	77
推薦閱讀.....	77

4 打碎數字：一般目的計算器..... 79

錯誤復原.....	81
資料結構.....	82
變數儲存.....	85
計算的主常式.....	87
全盤變數的初值化.....	90
三個簡易的模組.....	91
摘取格式指引.....	92
敘述語法.....	93
認識識別符號.....	96

表式求值.....	98
項目求值.....	100
因素求值.....	101
算 術.....	102
一個字串轉變成一個廣域實數.....	107
將廣域實數轉換成一字串.....	109
變數的儲存和檢索.....	112
建 議.....	115
推薦閱讀.....	117
5 本文檔案工具.....	119
選 替.....	119
APPLE PASCAL 本文檔格式	121
本文檔資料結構.....	121
打開輸入檔常式.....	123
建立新檔常式.....	125
關閉檔案.....	126
自本文檔讀一字串 tfread	127
組合語言.....	134
單元和節.....	141
度 量.....	143
排 印.....	145
全盤變數初值化.....	150
改變排印參數.....	151
取得檔案名稱.....	152

排印檔案.....	153
輸出初值和終止.....	154
解碼逸出順序.....	157
印各個檔案.....	159
字串和字元輸出.....	162
頁格式.....	163
找包含指引.....	164
建 議.....	167
推薦閱讀.....	167
6 遊戲與戰略.....	169
翻棋的規則.....	170
運轉程式.....	172
資料結構.....	173
翻棋的主常式.....	176
設定全盤變數的初值.....	177
顯示棋盤.....	180
簡易棋賽.....	180
定比賽變數的初值.....	182
雜項的簡單常式.....	184
找合法移動.....	186
取得玩家的着手.....	189
下手棋.....	190
結束棋賽.....	192
比賽理論.....	193

取得電子計算機的着手.....	197
估計一棋盤位置.....	201
將變棋表列排序.....	204
衡 量.....	205
APPLE 圖形初階	208
修飾 reversi 圖形.....	211
建 議.....	220
推薦閱讀.....	221
7 模擬與漫畫.....	223
資料結構——座標和向量.....	225
實 數.....	227
全盤變數定初值.....	230
取得參數常式.....	231
整數輸入.....	232
實數輸入.....	234
Stor 常式	241
runbouncer 常式	244
initbrun 常式	246
moveballs 常式	248
建 議.....	251
isaac 模擬.....	252
重力作用.....	256
計算的方法.....	258
更多的資料結構.....	259

推薦閱讀.....	265
8 電子工作表格.....	267
Pascalc 作業方式	268
資料結構——稀疏矩陣.....	271
Cell 記錄	274
格 式.....	276
定記憶大小.....	277
Pascalc 主常式	279
全盤變數定初值.....	285
Sheet 顯示常式.....	287
游標的放置與搬移.....	292
遞增測試.....	294
鍵入標記.....	294
未輸入.....	296
動態記憶配置.....	302
記憶體管理資料結構.....	304
APPLE PASCAL 記憶體管理	309
字串的動態儲字.....	313
Cell 儲存	314
公式儲存體.....	317
進入公式.....	318
重新計算.....	323
和函數.....	328
重定格式.....	333

命 令.....	334
簡單命令.....	336
顯示和改變格式的參數.....	338
抄的命令.....	342
抹除的命令.....	347
插入命令.....	354
載入與儲存.....	357
reary sales	363
排印表格.....	365
建 議.....	370
推薦閱讀.....	371
附錄 A 易傳性問題的探討與對策	373
字 串.....	373
長整數.....	382
集 合.....	394
隨 筆.....	396
其他問題.....	399
推薦閱讀.....	401

1

什麼是好程式

(*What Is a Good Program ?*)

什麼是一個好的程式？這個問題並不簡單。因為，我們並沒有可以用來衡量程式是「好」的方法。這不但包含了程式的設計，同時還牽涉到經濟、心理、和哲學。所以，在此我們並不打算給這問題一個答案，而是試著去檢討一下。

當我們談到一個程式的好與壞時，我們當然是指該程式的品質，而非指該程式的實際用處。「好」與「壞」這二字經常混淆不清，因為，它們是道德上的字眼。而電子計算機與程式，却完全不涉及道德問題，它們只純粹是個工具。但是，正如同一個鐵鎚可以用來建造醫院，也可以用來敲碎人的頭蓋骨，電子計算機同樣地可以用在好的或是用在邪惡事物之上。

這本書將不討論到道德哲學。即使是好的程式也可能被用於邪惡之途。我們仍然定一個比較「功利」的定義：一個程式的好，在於如何完成它的工作，而與該工作的好壞無關。

我們可以由兩個觀點來看程式的品質：從程式設計師和程式使用者的觀點來評判。使用者所關心的是程式的外表：從它的性能評估好與壞，以及處理錯誤能力等等。程式設計師則著重在它的內在：清晰性（clarity）、易傳性（portability）、和模組性（modularity）。在這兩種觀點中，使用者的

地位重要得多，因為程式的目的就在於日後使用它。有時候，程式使用者就是程式設計師本人，但是大多時候，程式設計師是為別人寫程式。無論所謂別人是指什麼樣的對象，我們得到的結論是相同的：一個好的程式是以使用的人為中心而寫的。否則寫一個程式，却不為使用的人所接受，則徒然浪費時間。

使用者的準則 (User's Criteria)

顯然地，一個好的程式必須能「工作」(work)，也就是正確無誤的達成設計的目的（實際上，某些數目和種類的錯誤仍然是可以接受的。）但是，仍有許多程式雖然可以工作，却不為使用的人所採納，或是根本就無法滿足使用者的需要。所以，除了程式是可以工作的能力 (workbility) 之外，我們還得增加一些「好的準則」(good criteria)。

下列各項準則，彼此都有些牽連：

有效性 (USEFULNESS)

是否要把即將寫的程式放在第一位？還是設計師先花些時間在別的工作上？這兩種何者比較具有生產力呢？這程式能滿足已知的或未來可能的使用者的需要嗎？這些問題看來很清楚，但是有許多的程式，却往往只能解決程式設計師自己心目中的問題。（本現象在個人用電子計算機界更為厲害。）相反地，有些程式除了被廣泛採用之外，甚至還可以解決一些連程式設計師自己都從來沒有想到的問題。

有效性通常都是由客觀條件來衡量的。例如，一個醫生的發帳單程式 (billing program)。因為比起以前的人工操作系統，現在正確的準時付款和未付款的資料等，使得他每月可節省 1489.34 美元。或者一個文字處理程式 (word processing program)，可以節省 53% 的文件準備時間。而不好

的程式却會增加成本。

程式的有效性，也可以部份地，或完全地以主觀因素來衡量。尤其是對於被設計用來遊戲、或娛樂方面使用的程式。

易使用性 (EASE OF USE)

電子計算機是一種能消除乏味的工作之工具。然而，為什麼許多程式仍要求使用者自己執行一些既費時、又容易出錯的工作？為什麼常用程式的特徵比極少用到的程式之特徵更難掌握利用？為什麼許多軟體註釋文件寫的雜亂無章？

在許多情況之中，造成程式使用不便的原因是由於程式設計師的懶惰。甚至包括程式設計師的許多人都認為：只要程式能作用，其他的不過是一些修飾門面的工作，無關緊要。

使用者對於一個程式的第一次接觸，是在他第一次使該程式時。這也是使用者建立起他對這個程式的態度的時候。學習通是一種艱辛，甚至於痛苦的經驗，學習如何使用一個程式也不例外。所以程式對於第一次使用的使用者，更應特別友善。

使用者不應該期待、或被要求一次就可以了解整個程式。尤其是對於一個很大、很複雜，並且有許多命令 (command)、及特性 (features) 的軟體。使用者最好試著不必先讀完一本很厚的文件 (document)，而能先使用手冊 (menu) 來操作的：在程式中很明顯的一些分段上，會有一列可供使用者自己決定的選擇機會，使用者只要直接按下一個或二個鍵即可。該使用者手冊可以列入自修課程內容之中，以便初學者按步就班的逐一測試，或者有些程式自己在內部解釋的很清楚，使用者根本不須參閱使用者手冊。另外有些程式內部含有特殊的指令 (instruction)：如一個「求助」 (help) 命令可以顯現出如何使用該程式的資料，或者解釋該程式的某些特徵 (feature)。

有的程式開頭非常易學，時間一久即嫌累贅。初學者很快

就覺得厭煩，而有經驗的老手更是失望。所以，設計一個能同時使初學者和老手都感到愉快的程式，是非常具有挑戰性的工作。有些程式也就因此分有「專家」和「初學者」的不同模式（mode），以滿足各個不同層次的使用者。

另外一個可以使程式易使用性增加的因素是一致性（consistency）。這只是保證使用者在使用同一程式的時候，不須去記憶一大串不同的規則（rule）。例如，在同一程式中，使用者只須回答是與否（yes or no），或者輸入數字的資料（numeric information），誤差訊息（error message）永遠以同樣形式、在同樣地方出現。同時，使用者在控制程式流程（flow）之時，能維持一定的步驟，也能達到一致性的目的。例如，統一使用〈逸出〉鍵（〈ESCAPE〉鍵）來強迫停止該程式現階段的處理（current process）而回到該程式的上一個主段落（previous major section）。

效率性（EFFICIENCY）

假如有兩個功能完全相同的程式，除了程式A比程式B的處理速度快了20%。人們會喜歡用那一個？再假設有兩個管理郵件目錄的程式，在同樣大的空間上，其中一個程式能儲存兩倍資料於其上，而不影響其他結果。同樣的，你會選擇那一個？

人們使用電腦是爲了使工作簡化，而且不費時。這個事實引導了效率二字的存在：在其他情況都相同的情況下，人們會選擇效率高的程式，因爲它能在等量時間內完成更多的工作，或是較短時間內完成工作。

但是程式設計師常常過份重視效率，以致於他想要去除任何不必要的、微小的代碼（code）和數元組（byte）以達到節省時間、或空間。這使得本程式代碼（program code）很難閱讀，甚至於根本無法操作。所以對於效率最適當的做法

，是把它當作一個值得的、應注意的素質，而不可忽視它、或太計較它。

伸縮性 (FLEXIBILITY)

通常，程式對於某一特殊事項都能處理的很好。很少程式能輕易的被利用到別的目的上。更少程式能配合，並且有效率的和別的程式一起工作。

一個具有適應能力的程式是可以滿足數種不同的應用方向的。例如一個本文編輯器 (text editor)，它同樣適用於程式本文 (program text) 和平常的散文 (prose)。另外一個例子是一個通用的儲存、檢索資料 (data storage and retrieval) 的程式，它能用來處理各種不同的資料，而不僅限於郵件。

另一種伸縮性可由程式中是否能根據使用者自己的需要或特質而修改。這樣的程式也允許使用者決定這個程式如何提出問題，接受資料以及設定輸出規格等等。一個伸縮性強的程式也應該可以利用到系統中特殊的硬體裝置。

程式的伸縮性有時候會造成程式有太多的特徵 (feature)。除非這些特徵能協力地、切實地、明智地工作在一起，否則會使程式喪失組織性，並且難學、難用。因此，一個真正伸縮性強的程式是能以一種簡單而又直接的方式，為多種環境所接受。

可靠度 (RELIABILITY)

為了許多不同的原因，即使在正常的情況之下，程式往往會出乎意料之外發生失誤。(根據「墨非定律」(Murphy's Law)，程式也許在不可能的情況下發生誤失。)一個令人無法信賴的程式，顯然會令使用者失望，特別是在原因不明的時候。而程式的不可靠，往往是在寫完之後才察覺到，這段時間的浪費，增加程式找出錯誤 (bug) 的麻煩。

最常發現導致程式不可靠的因素是沒有在程式中檢查輸入的資料。有時候，這個錯誤非常明顯。例如，程式要求使用者回答 Y 以表示 Yes 或者是 N 表示 No，而未能檢查使用者是否輸入其他的答案。再如，一個程式只能計算在某一區間內的數字，却未檢驗輸入的數字是否落在該區間之內。程式可能會因此失敗，甚至更糟的，提供了不正確的答案。雖然沒有一個程式能阻止使用者輸進錯誤的資料，但是應該要檢驗，以防止任何無心的錯誤，以及可偵察出的輸入錯誤。

可靠度的問題出在沒有好好計劃的設計。使得一個程式能防止失敗（crash-proof）是必須不斷地去測試任何種突發狀況：除數為零，讀一個根本不存在的檔案，溢位（overflow）或超下限（underflow）等等。有些很容易就能偵測出（如果除數是零），有些就很難、或根本不可能偵測出（例如嚴重的硬體問題。）

當然，只偵察這些不被期待的狀況，只是解決問題的一部份，程式設計師必須要決定如何去處理這些狀況，這通常更棘手，我們將會於後再討論，並舉些偵錯和處理的例子。

適用性（SUITABILITY）

這個準則對於不同的使用者，會有不同的條件。對於一個星期五下午送一個程式，而於下星期一早上才回來看結果的人，則效率就不是最重要的因素了。對於經驗老到的程式設計師所用的偵錯工具（debugging utility），易使用性的標準又要為秘書們做的文字處理的程式不同了；同樣的道理以三年級的學童為使用者的處理程式也不相同。所以，程式設計師首先必須知道，其設計的程式之使用對象。

當然，軟體可能是針對某一小群的人、或是廣泛的大眾而設計的。在其中的任何一個方向都可能會有陷阱。專為一小群人設計的可能是太專門化，使得其他的人無法使用。另外一方面

，把對象定的太大、太多，則程式依然無法使每個人都滿意。

程式設計師的準則 (Programers' Criteria)

我們曾討論程式設計師，應該關心他們所設計的程式是否能讓使用者滿意。但是，程式設計師必須了解，有些使用者既不懂，而且也不在意程式的清晰性 (clarity)、易傳性 (portability)、和模組性 (modularity) 等等。

如果，設計師能寫出完美無缺點的程式，他就不必擔心這些準則了。然而，程式通常是需要修正的，例如，他們的程式，有時候會移到不同的機種上使用，或是相同的機器而不同的軟體環境下使用。也有時候，使用者希望能再增加一項特徵，或取消某一項特徵。這些情況都會牽涉到程式的修訂。

因此，任何不屬於瑣碎、少用到的程式，在設計的時候都應該考慮到該程式爾後也許還會再使用，也許會再修改、再升級。其中最重要的是程式的再升級 (upgradability)，它必須是開放式的結束 (open-ended)：特徵的加或減都不需要大幅度的重寫。

另一個修改程式，以減少問題的數字的方法，尤其是在大的複式程式的大企劃，是要小心的為原始程式 (original program) 和隨後要改變的做文件 (documentation)，比方說是誰、何時、為何要改變？

除了要考慮這些要項之外，很重要的是程式盡可能要做到易於了解。即使負責修改程式的人就是當初撰寫該程式的設計師，他也許已經把程式的使用技巧和細節部份遺忘了。如果程式過於複雜或草率，都會造成日後修改的總總不便。

下列的各項準則，和使用者的準則一樣，彼此或多或少有些牽連：

可讀性 (READABILITY)

藉著程式代碼 (program code) 具有簡單的格式 (format) 、可以去了解程式。矩齒狀 (indentation) 可以表示出程式的結構。空行可以用來分開程式的各個不同的部份，空格則避免了程式敘述 (program statement) 間的狹窄限制。因為 PASCAL 語言對於根源程式 (source program) 的格式不設限制，程式設計師可以隨心所欲地採用自己的方式。

在本書中，我們將採用一個具有一致性的格式 (format) 規定。這並不武斷的表示該規定的價值，讀者們可自己發展自己的規則，只要前後連貫一致。這個特性將有助於程式碼 (code) 及邏輯上的錯誤偵測。

易傳性 (PORTABILITY)

一個易傳的程式不需經太多的修改，便能在其他計算環境 (computing environment) 下使用。修改的愈少，其易傳性就愈高。換句話說，可傳性通常指的是一個程式轉移到另外一個電子計算機的容易程度。它也是指在電子計算機的操作系統 (operating system) 的修改之下，或改變硬體結構、程式的生存狀況。

可傳性對於希望程式能被採用的設計師而言，十分重要，因為適用的機種愈多，使用者的範圍也愈廣。而可傳性對於希望能快速提高程式品質，減低硬體消費的情況影響更大。因為今天深受歡迎的電子計算機，日後還是會落伍的。可傳性高的程式，就有比較大的機會，在未來的電子計算機上使用。

清晰性 (CLARITY)

清晰性與可讀性 (readability) 十分相似。它指的不是程式本文 (program text) 實質上的格式，而是該本文是否能表達出程式設計師到底要完成什麼。就這一點而言，好的設計和好的寫作十分相似。

• 1 什麼是好程式 •

一個能使程式更清晰的主要方法是有意義的採用識別號 (identifier)。假設一個變數 (variable) 代表的是另一方程式的根，那麼它的名字以 **root** 表示遠比較用 **x** 要恰當的多，又比如在一個二分樹 (binary tree) 中插入點 (node) 的步驟中，將點的命名為 **insert node** 要強過 **do it**。程式中的識別號，必須要使其混淆的程式降到最低。例如，不要同時取下面兩個名字：**finddel** 及 **finddle**。一個受到大家歡迎的建議，是在不同的識別號的等級 (identifier class) 中，使用不同的詞性：動詞給程序 (procedure)，形容詞給布耳函數 (Boolean)，名詞則留給其他函數。

另外一種增加程式清晰性的工具是在程式中列入適當的註解 (comment)。註釋是非常有用的路標，可以為每一個代碼段落 (code segment) 的活動做總結，可以指出在某種情況下，該程式選擇那一條分支 (branch)，也可以特別說明程式中所需要的技巧、與較難處理的邏輯問題。尤其是一個大計劃，好幾個程式設計師必須要閱讀並修改別人的程式，那麼註釋就顯得格外重要了。因為沒有人知道，那個唯一懂得一重要程式段落 (program segment) 的人是否需要他時，都能到場。

使用 PASCAL 語言的設計師比較幸運，因為該語言鼓勵大家採用結構化規劃 (structured programming)、遞迴 (recursion)、指標變數 (pointer variable) 及新資料型 (data type) 的定義。如果明智的使用，都可加強程式的清晰性。

清晰性常被和冗辭 (wordiness) 混為一談，但他們的相反是一致的：簡明 (conciseness)。不要認為每五行就有一個註釋即能使程式看來更明白。它只會徒增程式的不可讀性，因為它妨礙了程式的邏輯。(如果一個程式的邏輯糟到需要大

量的註釋來補充說明，那麼在加入註釋前，即先改進程式本身。）

同時，有意義的識別號（identifier）也不必太長，以免令人生厭，也易造成錯誤（bug）。因為 PASCAL 語言只接受前八個字元（character）來區分識別號，造成有時候兩個以上識別號，會代表一個變數。幸運時，只是造成編輯錯誤（compiler error），甚至於它會引起無法收拾的後果。

模組性（MODULARITY）

將一個程式分成若干較短、較易被了解的模組（module），對於提高程式的品質貢獻很大。（請注意，我們使用模組這個字，同時這個字代表 PASCAL 的程序及函數。但是，如何去分段才是對程式設計師最有利呢？

一個好的模組將可正確無誤地完成一項任務（task），不多也不少。它藉著簡單的通訊路徑，例如幾個變數與呼叫常規（calling routine）溝通訊息。這樣類似的模組使得程式設計師不需要一次完全了解所有的代碼（code）。因此，他們可以直接先研讀，甚至修改程式中的某些部份，而不必要審查或重寫其他的地方。這對於除錯（debugging）和修改程式，均有很大的幫助。

對於想要把程式組合在一起，模組性也能達到節省規劃時間的效果。一般目的之模組（general-purpose module）往往可以在不同的程式中使用，這避免了不必要的重複與浪費。主要的關鍵就是在保持模組的伸縮性（flexibility），使之適用於各種不同的情況，而不是僅針對第一次使用到這個部份的情況。

程式語言（Programming Language）

• 1 什麼是好程式 •

程式語言 (Programming Language) 的本身就是程式，我們可以用前面提到的使用者準則來評估它。這時，程式設計師就成了使用者。例如，這種語言是否能處理它本身的簡單簿記工作 (book keeping)。還是它需要程式設計師自己費一大番功夫？在設計師失誤的時候，它是否能提供有用的誤差訊息？一個程式大小 (size) 的限制是多少？如果這語言是做為編譯器 (compiler) 的，它編譯程式的速度有多快？它是否允許設計師應用到硬體上的特徵或是操作系統的能力？是否仍有些事情電腦處理，而這種語言却不可以？（你可以自己加上許多問題，只要你稍動腦筋。）

更重要的是，好的語言應儘量使得寫好的程式輕而易舉。一個程式設計師，不該在語言所造成的限制上浪費時間。

在現實世界上沒有一種語言是十全十美，也可能將來也無法發展出來，因為沒有某一種單獨的語言可以滿足各種情況，程式設計師可能是人工智慧的研究人員，電子計算機的喜好者、工程師、或者是經濟學家，他們的要求各不相同。

在下面若干章內，我們會發現選擇 PASCAL 語言設計程式是明智的；因為它具備了結構化和易於了解的優點，但無論是 PASCAL 語言本身，或是實施在 APPLE 上的 PASCAL 它們都有一些缺點。

在後面我們會看到，我們有時候並不用 PASCAL 去做我們想做的事，而是「誘使」 (tricking) PASCAL 去做，把它們視為事實是合理、實用的態度。

推薦閱讀 (Recommended Reading)

由 Brian Kernighan 和 J.P. Plauger 所著的 The elements of programming style 應該為程式設計師所必備的。

• 高等 Pascal 程式設計技巧 •

該書中錄有一些好好壞壞的程式，並說明好的基本法則，其撰寫方式相當吸引人。

Marilyn Mehlmann (Prentice Hall 1981) 的 When People Use Computers 一書，討論如何寫出讓人容易使用的程式。

而 Joseph Weizenbaum 著的 Computer Power and Human Reason 則有關於如何適當的使用電子計算機技術的報告，十分有趣，可看性高。

2

陰極射線管技術

(*CRT Techniques*)

根據前面所論，我們已知道，評估一個程式品質的最重要準則是這個程式如何與使用者溝通作用。對大多數使用者而言，這個溝通泰半是藉著映象顯示器（video display），也就是陰極射線管（CRT）達成的。雖然陰極射線管（CRT，或 Cathod Ray Tube）指的是射線管的本身，但我們將用它來代表電子計算機的映象顯示系統（video display system）。

有效的使用陰極射線管的螢幕（screen），可以使得程式更好使用，並且更顯得專業化。幸運地，利用陰極射線管的特徵對於 PASCAL 程式設計師而言，是非常容易辨到的。

不久之前，利用陰極射線管來作為交作式電子計算機（interactive computer）的工具是一項相當花費的奢侈行為。大多數人只能被限制在「整批處理」（batch）的環境之中，與電子計算機之交作（interactive）關係幾乎沒有。即使是早期的交作性電子計算機，比如共時電子計算機（timesharing mainfram machine 和迷你型電子計算機（mincomputer）也只能主要經由印字終端機（printing terminal）與電子計算機進行「交談」。

今天的映象顯示系統，要比印字終端機來得快多了，也便

宜多了。而電子計算機硬體成本的降低，更使得交作性電子計算機要比非交作性電子計算機普遍。因此，如果你發現有以印字終端機為主要輸出裝置的個人電腦出售時，一定感到驚訝的。他們多少都會有類似映象顯示系統的。例如，個人電子計算機可以有內建陰極射線管（built-in CRT），連接到電視機或映象監察器（video monitor）的介面設備（interface），或是連接到獨立陰極射線管終端機（stand-alone CRT terminal）的介面。

由於低價格陰極射線管技術的發展，我們現在有了一種新型的電子計算機軟體，姑且稱之為「螢幕導向」（screen-oriented）的軟體。螢幕導向的程式與其他程式不同在於他們需要映像顯示，藉以達成工作。特別是，他們藉著陰極射線管的能力，去放置資料於螢幕上。他們也可以利用螢幕的強光效果（highlighting）、不同字元組（character set）、甚至於聲音。相反地，非螢幕導向（non-screen-oriented）的軟體是為也能在印字終端機上操作而設計的。

螢幕導向的軟體有過於冗贅的傾向，因為它不斷地隨著程式的狀況為使用者提供消息、數據。這些反應可以幫助使用者更有效地使用程式。相反地，非螢幕導向的程式則傾向於儘可能地減少提供這種輸出給使用者，而希望使用者能不依賴它而正常操作程式。畢竟，沒有人願意花時間去等印字終端機吵雜地一行行印出那些消息，何況那些消息尚不一定都有用呢！這些非螢幕導向程式又稱為「窄寬帶」（low bandwidth），以表示他們所能給使用者的反應有限。因為要能夠便捷地與程式交通的情況下，使用螢幕導向的軟體，就比非螢幕導向的軟體來得實際。

所以受歡迎的陰極射線管可以在程式的命令（command）控制之下，清除整個或部份螢幕，把游標（cursor）移到某一

特殊定點，或是可以任何方向地調整游標。除了這些基本功能之外，有許多高級的作業也常被提出。例如，使螢幕上的資料能做上、下、左、右的捲移（scroll），或在螢幕上形成各個獨立的窗框（window）等等。有的陰極射線管也能提供一些像是否要讓字元閃爍（blinking）、反向顯示（inverse-video），以及不同字元組（如大、小寫）等等的選擇。最後，許多陰極射線管還能提供不同的繪圖能力，由簡單的圖號組（graphics character set）乃至於複雜的全色線條（full-color line）和圖形（figure）。

任何兩個不同廠牌的陰極射線管所具備的功能各不相同。即使他們執行相同的功能，他們使用的指令也不一樣。所以，螢幕導向程式的缺點之一就是缺乏易傳性（portability）。現在雖然有美國國家標準協會（ANSI：American National Institute），可以提供某些功能，但這並不完全為大家所採納。因此，一個程式可能在某一電子計算機和陰極射線管的組合之下，表現優異，却可能在另一種情況下完全失效。譬如在A型射線管上清除螢幕的命令，可能會導致B型射線管胡說八道。所以一個螢幕導向的程式，必須先顧慮到各種不同型的陰極射線管，或是藉由一個與陰極射線管性質有關的已設定步驟（CRT-dependent installation program）來修正這程式，才可予以使用。

利用陰極射線管的特色（using a CRT'S Special Feature）

讓我們試用PASCAL語言，寫一個螢幕導向程式的可能方法。首先，我們挑一些比較常用的陰極射線管方程式來做。接著，我們給一系列可以利用映像顯示器操作的功能取一個名字：crtcommand，來代表一些PASCAL中螢幕和游標的指令如下：

```
<type>
  crtcommand = (HOME, CLEAR, ERASEOL, ERASEOS, UP, DOWN, LEFT, RIGHT, BEEP);
```

< type > 是用來表示 crtcommand, type 這個宣告部份 (declaration section) 中定義。我們在下面的表 2 - 1 中, 可以看到每一指令的解釋。

表 2-1 陰極射線管指令

命 令	結 果
HOME	將游標移到螢幕左上角。
CLEAR	清除螢幕, 再將游標移到左上角。
ERASEOL	將同一列內游標現行位置及其後位置完全清除。
ERASEOS	將螢幕上游標現在位置及以後完全清除。
UP	游標向上移一行。
DOWN	游標向下移一行。
LEFT	游標向左移一格。
RIGHT	游標向右移一格。
BEEP	使終端機或電子計算機發出「嗶」的聲音。

下一步, 我們利用這些指令寫一個程序 (procedure) 名為 crt。例如 crt (up) 這個敘述 (statement) 使得游標向上移一行。大多數的終端機 (terminal) 上, 這些指令都非常容易使用。事實上, 除了我們所列出的這些簡單功能之外, 許多終端機都有其他的特殊作用。當然, 我們會碰到一個可能的問題。crt 隨著不同的種類之陰極射線管而不同。通常, 針對某一特殊作用。當然, 我們會碰到一個可能的問題, crt 會隨著不同種類的陰極射線管而不同。通常, 針對某一特殊的陰極射線管功能來寫這個 crt 比較直接的方法。主要的障礙就是

去查詢並譯釋 (decipher) 有關這個終端機、或操作系統的資料。下面是一 APPLE PASCAL 的實例：

```
procedure crt(cc: crtcommand);
( Do CRT command, Apple Pascal version )

begin ( crt )
  case cc of
    HOME:
      write(chr(25));
    CLEAR:
      write(chr(12));
    ERASEOL:
      write(chr(29));
    ERASEOS:
      write(chr(11));
    UP:
      write(chr(31));
    DOWN:
      write(chr(10));
    LEFT:
      write(chr( 8));
    RIGHT:
      write(chr(28));
    BEEP:
      write(chr( 7))
  end
end;
```

這個針對 APPLE II 的 crt 程序是利用案例 (case) 來處理每一個指令的反應。不論是那一種案例，他們都會送出一個字元：如果是 crt (down)，我們得到 <LF> (即 ASCII 10 , Line feed)，如果是 crt (ERASEOL)，得到 <GS> (即 ASCII 29 , group separator) 等等。這個案例的結構，非常簡單而清楚。如果不只是送出簡單的字元，我們希望得到一些比較複雜的反應，那麼我們只要在相對應的案例標記 (case label) 下，加入適當的代碼 (code)。

在這個程序設計中，我們做了一個不太明顯、但可能是不具易傳性 (non - portable) 的假設。換言之，也就是認定內建程序 (built - in procedure) 的 write 存在。我們假定 write 的實施，可能要使得這個被等待輸出的本文 (text) 先在輸出緩衝器 (output buffer) 中等待，等到該緩衝器滿了、或已

累積到了一整行 (an entire line) 的本文，才把它們送出去。雖然標準的 PASCAL，並不特別使用緩衝器的輸出方式，但也沒有禁止它。所以，如果你的 PASCAL 採用輸出緩衝器，那麼就不該使用 **crt** 中的 **write**，而要改用另一個可以使得輸出立刻被執行的常式 (routine)。這可能是呼叫一個操作系統，或緊跟著排除緩衝器 (buffer flush) 的 **write** 動作。

另一個必須考慮到螢幕導向的程序，是將游標移到螢幕上某一個定點。這種程序已經被建立在APPLE和UCSD PASCAL之中，稱為 **gotoxy**。呼叫到 **gotoxy** (**x** , **y**) 的程序會使得游標停留在螢幕上第 **x** 行、第 **y** 列的位置上。我們通常把螢幕的左上角位置定為第 0 行、第 0 列。行數向右遞增、列數往下遞增。在一個 80 行、24 列的螢幕上，最右下角的位置就該是 79 行、23 列了，這個 **gotoxy** 程序將在本書中出現，因為螢幕導向的程式來說，它占的分量十分重要。

如果某一版的 PASCAL 並不包含 **gotoxy** 為一個已內建的程序，我們可以簡單地自己加進去，下面有一個實例：

```
procedure gotoxy(col, row: integer);
  ( Send cursor to given position, Z-19 version )
begin ( gotoxy )
  write(chr(27), 'E', chr(32 + row), chr(32 + col))
end;
```

上面的例子是假定使用 Zenith Z-19 型的終端機。對於別種型式的陰極射線管，只需要得到它的硬體手冊 (hardware manual)，再作修正即可。

所有在 **gotoxy** 中出現的、而且與陰極射線管有關的事實上都是 **write** 這個敘述。我們可以發現，Z-19 會送出 4 個字元，來做陰極射線管的定位：<ESC> (ASCII 27)，<E> (ASCII 69)，和另外兩個由 32 分別加上行數、列數所得到的美國資訊交換標準字元 (ASCII Character)。

一旦 **crt** 和 **gotoxy** 這兩個單元寫好了，我們在程式中都得藉呼叫這兩個常式來完成一些工作。這個使得所有與陰極射線有關的軟體都受制於這兩個單元。如此一來，易傳性的問題就降低了：因為一個程式移到另一終端機時，只要改兩個常式即可。

現在，我們來看參數（Parameter）的檢驗。在上面的 **gotoxy** 中，我們假設得到的參數是有效的：列和行的數目可以對應到一個存在的螢幕位置。同時，我們也有考慮到 **crt** 該如何處理一些螢幕的極端位置（**extreme situation**）的狀況。例如，游標已經在最下一行，而程式接到 **crt**（**DOWN**）的指令時，該怎麼辦呢？根據硬體的設計，整個螢幕可往能上捲一行，或是將游標移動到螢幕的第一行，也可能什麼也不做。

藉此討論，我們要指出 **crt** 和 **gotoxy** 本身，並不負責處理不正確的情況或參數。這與好或壞的程式設計之策略無關，只是一個在設計時限單純的決定，讓呼叫 **crt** 或 **gotoxy** 的常式自己負責。（也許，你會想到如果我們做的是另一種決定。**crt** 和 **gotoxy** 該如何改變？如果 **APPLE** 和 **UCSD PASCAL** 中，**gotoxy** 是一個內建程序，該找出當它收到錯誤的參數時候，它將如何。）

crt 和 **gotoxy** 這兩個模組的工作，就像是堆積木一般地，一步一步完成複雜的工作。例如，下面的敘述會使得螢幕先被清除，接著在 10 列、5 行處出現 **<Hello there!>** 的字串：

```
crt(CLEAR);
gotoxy(5, 10);
write('Hello, there!');
```

使用這種方法來執行陰極射線管功能的主要好處，在於如此。我們可以藉著前面提到的基本作用，輕鬆地達到比較高級的技巧。假設某台終端機顯示反向（白底黑字）字元，在 **crt**

command 中的宣告裡，可以加入兩個指令：

```
<type>
  crtcommand = (HOME, CLEAR, ..., BEEP, INVON, INVOFF);
```

那麼，**crt** 程序修改如下（我們還是假設使用 Z-19 的終端機）：

```
procedure crt(cc: crtcommand);
...
  case cc of
    HOME:
...

    INVON:
      write(chr(27), 'p');
    INVOFF:
      write(chr(27), 'q')
...
end;
```

此時，呼叫 **crt (INVON)** 會使得反向字元作用生效，而 **crt (INNOFF)** 則使其失效。

crt 一個令人挑剔的地方是它缺乏效率。或許有人會提議，爲何不把送出的控制字元（control characters）建議成一個陣列（array），在程式的一開始，該陣列設定初值，而不必用到 **case** 敘述。如此 **crt** 只要檢查字元順序、並將之輸出即可，讀者可以試用此法，並比較看那種情況較快。

另外尚可以利用設定初值化的常式（initialization routine）把與陰極射線管硬體相關的資料，輸入程式之中，而完成 **crt** 和 **gotoxy** 的工作。這是利用程式本身必須的易傳性（portable），而留有一個資料檔（data file）會因爲硬體不同，而改變內容。許多這樣的技巧被利用在 APPLE 和 UCSD 的 PASCAL 系統之中。有關 **crt** 和其他硬體的資料都被存在

• 2 陰極射線管技術 •

一個稱之為 SYSTEM 的檔案。當系統被靴式啟動 (booted) 時, 則 MISCINFO 被讀到記憶體中, 系統程式就利用這些資料去執行與硬體有關的工作, 例如, 清除螢幕 (Screen - clearing)。(這個方法的優點是什麼? 是不是有什麼壞處?)

```
program theseus;
( Create and solve mazes on CRT )

uses applestuff; ( for "random" and "randomize" )

const
  MAZECOLS = 38;      ( Maximum maze column number )
  MAZEROWS = 22;      ( Maximum maze row number )
  MAXCRTCOL = 39;     ( Maximum CRT column number )
  MAXCRTROW = 23;     ( Maximum CRT row number )
  XINDENT = 0;        ( Number of columns to indent maze )
  YINDENT = 0;        ( Number of rows to indent maze )

type
  mazesquare = (WALL, PATH);
  mazearray = array [0..MAZEROWS, 0..MAZECOLS] of mazesquare;
  crtcommand = (HOME, CLEAR, ERASEOL, ERASEOS, UP, DOWN, LEFT, RIGHT, BEEP);
  direction = UP..RIGHT;

var
  maze: mazearray;
  won: boolean;
  ch: char;

(-----)
( Modules to be included here: )
(      crt                      )
(      dispsquare               )
(      createmaze               )
(      solvemaze                )
(-----)

begin ( theseus )
  randomize;
  repeat
    crt(CLEAR);
    createmaze(maze);
    gotoxy(0, MAXCRTROW);
    write('Press <C> to continue:');
    read(ch);
    won := solvemaze(maze);
    gotoxy(0, MAXCRTROW);
    write('Press <C> to continue, <Q> to quit:');
    read(ch)
  until ch in ['Q', 'q'];
  crt(CLEAR)
end.
```

theseus 常式

現在，讓我們利用已經學到的工具來寫一個程式。我們要設計一個叫做 **theseus** 的程式來簡單地在陰極射線管上建立一個迷宮，並且解決這個問題。（**theseus** 是以一個希臘神話研究者命名。他成功的通過了一個大的迷宮之後，殺死了可怕的怪獸——人身牛頭怪：Minotaur。）這個選擇或許對某些人看來並不太重要，下面若干章中、將討論比較實際的程式。

theseus 的主程式如上：

下圖（2-1）即是經由 **theseus** 所計出來的迷宮。

因為，這是我們的第一個程式，我們將仔細的討論之。第一行敘述 **program theseus**；，只是給本程式定一個名字。其他版本的 PASCAL，可能還會要求列出這個程式使用到的所有檔案，而形成：**program theseus (input , output)**；

第二行，**uses applestuff**；，是通知編譯器（compiler）一個分離的編譯單位（unit）會被使用。這是 APPLE 和 UCSD PASCAL 的特性，使得一些經常會用到的常式，可以直接聚集在一起，而不需要再次重新編譯。**applestuff** 單位是 APPLE PASCAL 系統的一部份，**theseus** 只會使用到 **applestuff** 中的兩個常式：**random**（能提供一個虛擬整數（pseudo-random integer），該值介於 0 和 32767 之間）和 **randomize**（能產生不同的虛擬隨機順序（psedo-random sequence））。如果某一版的 PASCAL 未能提供內建的 **random** 和 **randomize** 常式，那麼就由程式設計師自行加入。在附錄 A 中提供了一個可行的方法：

theseus 所造出的迷宮，是由許多矩形的方形陣列（rectangular arrays）存在陣列 **maze** 中而得的；每一個矩形可能

```

XXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXX
X      X      X      X      X      X      X
XXX XXX XXX XXXXXXXXXXXX X XXXXXX XXX X X
X      X      X      X      X      X      X
X XXX XXX X X XXXXXX XXX X X X XXX X X X
X      X      X      X      X      X      X
X XXXXX XXXXX X XXXXX XXXXXXXX X X X X X
X X      X      X      X      X      X      X
X X X XXX X XXX XXX X X XXX X X XXX X X
X      X      X      X      X      X      X
XXX XXX XXXXX XXX X XXXXX X X XXXXXXXX X
X X X      X      X      X      X      X      X
X X X XXX X XXX X XXXXX XXXXXXXX X X XXX
X X X X      X      X      X      X      X
X XXXXX X XXXXXXX XXX X XXX XXXXX X X X
X      X      X      X      X      X      X
XXX XXX XXX X X XXXXX XXX X X XXXXXXXX X
X      X      X      X      X      X      X
X XXXXX XXXXX X XXXXXXXXXX X XXXXX XXX X
X      X      X      X      X      X      X
XXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

```

XXXXXXXXXXXX!XXXXXXXXXXXXXXXXXXXXXXXXXXXX
X      X      X      X      X      X      X
XXX XXX XXX!XXXXXXXXXXX X!X!XXXXX XXX X X
X      X      X X!X!-----!X X!X!X!X!X X X X
X XXX XXX X!X!XXXXXX!XXX!X!X!X!XXX X X X
X      X      X!X!-----X!X!-----!X!X X X X
X XXXXX XXXXX X!X!XXXXX!X!X!XXXXXX!X X X X
X X      X      X!X!-----!X!X!-----!X!X X X X
X X X XXX X XXX XXX!X!X!XXXX!X!X XXX X X
X      X      X      X!X!-----X!X!X!X      X X
XXX XXX XXXXX XXX X!X!XXXXX X!X!XXXXXXX X
X X X      X      X X!X!-----!X!X!-----!X!X X
X X X XXX X XXX XXX!X!X!XXXX!XXX!X!X X X X
X      X      X      X!X!X!X!X!X!X!X!X!X X X X
X XXXXX X XXXXXXXX!X!X!X!X!X!X!X!X!X X X
X      X      X!X!-----!X!X!-----!X!X X
XXXXX XXX X X!XXXXX!XXX X!X X XXXXXXXX X
X      X      X X!X!-----!X!X!X!X      X X
X XXXXX XXXXX!X!XXXXXXX!X XXXXX XXX X
X      X      X!X!-----!X!X!-----!X!X X
XXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

是WALL或PATH。要走出這個迷宮，電子計算機就必須要找出一條連續的PATH正方形，以便從迷宮邊境的一點、到達

對面的另一點。至於迷宮大小則是由兩個常數宣告 **MAZECOLS** 和 **MAZEROW** 而控制；它們被設計好，使得迷宮的大小、可以放進 **APPLE** 的 40 行 × 24 列的螢幕上。（如果要放到 80 行的陰極射線管上，**MAZECOLS** 可以增加到 78。）

MAXCRTCOL 和 **MAXCRTROW** 是用於存放陰極射線管螢幕大小的常數。**MAXCRTCOL** 是每一行陰極射線管上，所能容許出現字元的數目減一，請務必注意，我們由 0 開始計算的。同樣地，**MAXCRTROW** 則是每一列存放的字元數減一。此處，我們還是 40 × 24 的顯示器為主，如果欲用在 80 × 24 上，**MAXCRTCOL** 等於 79。

XINDENT 和 **YINDENT** 控制了迷宮出現的位置。迷宮的左上角端點，將出現 **XINDENT** 行和 **YINDENT** 列上。

在宣告的型態之中，**direction** 指出它的內容為上一行 **crtcommand** 中的一部份，共用的四個方向是：**UP**，**DOWN**，**LEFT** 和 **RIGHT**。這種表示比重新宣告一次來得方便。

方塊的註釋：

```
{-----}
{ Modules to be included here: }
{      crt                      }
{      dispsquare                }
{      createmaze                }
{      solvemaze                  }
{-----}
```

提出了所列出的程序將出現的位置和順序。在最後的程式之中，根源本文（**source text**）將出現在這方塊之中，並依照這個次序，它們就是一個個的模組（**module**）。模組本身內部可以再包含模組，這將由模組內的方塊註釋表示。由 **APPLE PASCAL** 所提供的常式，如 **gotoxy** 和 **random** 等，我們並不打算列出，並將視為「全盤」（**global**）內建的常式，供任何程式使用。

在本書之中，方塊註釋將是一種標準方法來取代。(1)：一整個程式、或是(2)：可由PASCAL編譯器或其他公用程式所提供的「包含」機構(include mechanism)。

現在，我們來看看實際的代碼。**theseus** 首先呼叫 **createmaze** 常式，以便在螢幕上建立一個迷宮。接著程式會令螢光幕上出現 **press < c > to continue**：的訊息。但使用著按了鍵後，**theseus** 才呼叫 **solvemaze** 來解本迷宮。

solvemaze 這個常式會回轉一個布耳(booleau)的值，以指出這個迷宮是否被解了。在我們的例子之中，**createmaze** 所建立的迷宮都有解答。所以 **solvemaze** 所回轉的值都該為 **TRUE**。既然如此，我們為何要它回轉一個值呢？因為，也許有一天，程式會被改變成可能有無解的迷宮。我們只是為了減少日後修正的工作。

等到迷宮有了解答，螢幕上會出現：**press < c > to continue, < Q > to quit**：。按下<Q>鍵，程式即停止，按下<c>鍵則有一個新的迷宮和解答。

值得檢討的是，使用者的按鍵回答，是這個程式較弱的一環，因為它並沒有偵查是否有效輸入資料。例如：如果在使用人讀到 **press < c > to continue**：之後，按下<RETURN>鍵，那麼迷宮整個就會往上捲移一行(**scroll up**)而遭到破壞。這個情況，在我們加入一些交互性的輸入工具時，可以得到改善，下一章將詳細討論之。

建立迷宮 (Creating the Maze)

讓我們看看如何設計 **createmaze**，以建立迷宮，為了吸引使用的人，我們建立迷宮的過程顯示在螢幕之上，這個可以輕易地由顯示 **maze** 陣列而達到。如果陣列中的元素是 **WALL**

，在螢幕相關的位置放一個<X>，如果元素是 **PATH**，即留下空白。

我們的策略是首先建立一個實心的迷宮（**solidrock**），也就是由 **WALL** 方塊組成。接著，再經由設定適當的元素為 **PATH**來挖洞（**excavate**），也就是我們先在迷宮中任意挑選一點，一個方向、開始掘出一個路徑。就用這種隨機的方法，使得這路徑曲曲折折，提高趣味性，直到不衝破邊界，不進入原先已挖好的路徑的情況下停止。最後，在迷宮的邊境上，找出「出口」和「入口」。

我們已經可以由上面非正式的敘述中，知道如何去寫這個 **createmaze**。但是，有時候利用這非正式的敘述，和正式的虛擬碼（**pseudo-code**）來輔助編寫、更要容易。在複雜的程式中，這是一個很好、很有效的技巧，因為它提供了一個中間過程，讓設計師可以組織他們的思想。這使得一大堆的邏輯設計、和偵錯的工作。在程式一行都沒有正式寫出的情況下，可以先在虛擬代碼的階段做到。

虛擬代碼有什麼特殊規則，它可不是一個組織上的技術。代表我們上面討論有關 **createmaze** 的虛擬代碼如下：

```
begin
  set maze squares initially to WALLs
  pick random point within maze
  set it to a PATH
  pick a random (legal) direction
  build maze path from that random point in random direction
  put entrance & exit on maze boundary
end
```

藉虛擬代碼，我們可以用 **PASCAL** 寫出這個常式。這處理涉及到將每一步的虛擬碼、換成數個 **PASCAL** 敘述，其中還包含了呼叫一些待會兒才會設計好的常式。下面即是轉換後的結果：

• 2 陰極射線管技術 •

```
procedure createmaze(var maze: mazearray);
{ Create a maze }

var
  row, col: integer;
  dir: direction;

{-----}
{ Modules to be included here: }
{      setsquare      }
{      rnd      }
{      randdir      }
{      legalpath      }
{      buildpath      }
{-----}

begin { createmaze }
  for row := 0 to MAZEROWS do
    for col := 0 to MAZECOLS do
      setsquare(row, col, WALL);
    row := 2 * rnd(0, MAZEROWS div 2 - 1) + 1;
    col := 2 * rnd(0, MAZECOLS div 2 - 1) + 1;
    setsquare(row, col, PATH);
    repeat
      dir := randdir
    until legalpath(row, col, dir);
    buildpath(row, col, dir);
    col := 2 * rnd(0, MAZECOLS div 2 - 1) + 1;
    setsquare(0, col, PATH);
    col := 2 * rnd(0, MAZECOLS div 2 - 1) + 1;
    setsquare(MAZEROWS, col, PATH)
  end;
end;
```

這個 **createmaze** 常式的結果，已經與我們的描述相當接近。但是仍然須要加入一些細節問題，有關迷宮中最早的隨機起點必須是放在奇數行、奇數列上（為什麼？），以及入口和出口，是選在迷宮的上、下二方。（事實上，在迷宮邊界上，

```
procedure setsquare(row, col: integer; val: mazesquare);
{ Set maze square to given value }

begin { setsquare }
  maze[row, col] := val;
  case val of
    PATH:
      dispsquare(' ', row, col);
    WALL:
      dispsquare('X', row, col)
  end
end;
end;
```

任何相異的兩點都可以成爲出、入口。)

大部份的真正工作，都是在 **createmaze** 中處理的，它們分別是 **setsquare**，**rnd**，**randdir**，**legal path**，和 **build path**。其中最基本的是 **setsquare**，它決定了迷宮中元素的位置是 **WALL**，或者是 **PATH**：

除了指派一個方塊的值之外，**setsquare** 呼叫了 **dispsquare**，以使得該方塊可以在螢幕上顯示出。如果它是 **WALL**，就出現 **X**；它是 **PATH**，則顯示出一個空白。由於迷宮裡的所有值都是由 **setsquare** 指派，因此這樣的安排，是爲了確保每一個迷宮的步驟，都可以在螢幕上顯示，下面是 **dispsquare**：

```
procedure dispsquare(ch: char; row, col: integer);
{ Display specified character in maze square }

begin { dispsquare }
  gotoxy(col + XINDENT, row + YINDENT);
  write(ch)
end;
```

createmaze 也呼叫了 **rnd** 這個函數 (function)，藉以找出迷宮中的隨機位置。**rnd** 產生出一個在一定範圍之內的隨機函數。

```
function rnd(low, high: integer): integer;
{ Return random number between low and high }

begin { rnd }
  rnd := low + random mod (high - low + 1)
end;
```

這個隨機整數的值、處於 **low** 和 **high** 之間，可能就是 **low** 或 **high** 本身。(在 **APPLE PASCAL** 中的 **random** 回轉的隨機整數介於 0 和 32767 之間。)我們把 **rnd** 寫成一個一般用途的常式，當我們需要任何介於兩數之間的隨機整數時，都可以使用它。純理論家可能會提出反對，因爲 **rnd** 並不能一直提供

• 2 陰極射線管技術 •

均勻分佈的數目（爲什麼不呢？）,但是此處 **rnd** 已經能滿足我們的目的了。

createmaze 另外又呼叫了 **randdir** 常式、藉之決定一個隨機方向；**randdir** 將由 **rnd** 得到的一個隨機整數轉換下列的四種方向之一：

```
function randdir: direction;
( Return random direction )

begin ( randdir )
  case*rnd(1, 4) of
    1:   randdir := UP;
    2:   randdir := DOWN;
    3:   randdir := LEFT;
    4:   randdir := RIGHT
  end
end;
```

要檢驗從某一點向某一方向延伸的路徑是否爲合法的（**legal**），是由 **createmaze** 呼叫 **legalpath** 可得知的：

```
function legalpath(row, col: integer; dir: direction): boolean;
( Return whether a legal path can be built )

var
  legal: boolean;

begin ( legalpath )
  legal := FALSE;
  case dir of
    UP:
      if row > 2 then
        legal := (maze[row - 2, col] = WALL);
    DOWN:
      if row < MAZEROWS - 2 then
        legal := (maze[row + 2, col] = WALL);
    LEFT:
      if col > 2 then
        legal := (maze[row, col - 2] = WALL);
    RIGHT:
      if col < MAZECOLS - 2 then
        legal := (maze[row, col + 2] = WALL)
  end;
  legalpath := legal
end;
```

前面曾提到過，如果一條路徑超越了迷宮的邊界、就是不合法，同時，它也不能破牆而進入一條已經建立好的路徑。

legal path在確定這兩種情況之後，回轉合法路徑一個**TRUE**值，如果是不合法的路徑，則回轉一個**FALSE**。

createmaze最後呼叫**build path**常式，這才是真正的挖洞工作。它先由一定的點，向一定的方向做路徑延伸，再試著由新到達的地點向外延伸，這是由神奇的遞迴（**recursion**）——**buildpath**，不斷地呼叫自己，以建立新的**path**。

同樣地，我們也可以先試用虛擬代碼來代表這個路徑過程：

```
begin
  extend path from specified point in specified direction
  for each possible direction from new path head
    check whether a path is legal
    if it is, extend the path in that direction
  end-for
end
```

除此之外，我們要使得新方向的檢查採取隨機次序（**random order**）；來確保迷宮本身的隨機性。下面即是用**PASCAL**寫的**buildpath**：

buildpath以隨機順序去建立四個可能方向，變數組**unused**只是暫時保存尚未測試的方向。等到**unused**空了的時候，即知已將各種方向試完了。

這些是我們用來建立迷宮的全部的東西了。這種演算（**algorithm**）確保迷宮充滿了路，而其中只有一條路可通達迷宮中的任意兩點（迷宮中不會有迴路（**loop**）出現。）。當然，更改了這個演算法，自然也就更改了這些迷宮的性質。

• 2 陰極射線管技術 •

```
procedure buildpath(row, col: integer; dir: direction);
{ Extend path in given direction }

var
  unused: set of direction;

begin { buildpath }
  case dir of
    UP: begin
      setsquare(row - 1, col, PATH);
      setsquare(row - 2, col, PATH);
      row := row - 2
    end;
    DOWN: begin
      setsquare(row + 1, col, PATH);
      setsquare(row + 2, col, PATH);
      row := row + 2
    end;
    LEFT: begin
      setsquare(row, col - 1, PATH);
      setsquare(row, col - 2, PATH);
      col := col - 2
    end;
    RIGHT: begin
      setsquare(row, col + 1, PATH);
      setsquare(row, col + 2, PATH);
      col := col + 2
    end
  end;
  unused := [UP..RIGHT];
  repeat
    dir := randdir;
    if dir in unused then begin
      unused := unused - [dir];
      if legalpath(row, col, dir) then
        buildpath(row, col, dir)
      end
    end
  until unused = [];
end;
```

解答迷宮 (Solving the Maze)

解答迷宮所需要的道理基本上，和建造它的時候一樣。要完成解答的程序之需求其實很簡單：它由迷宮邊界一端的起點、一直找出連續的 **PATH** 方塊，以達到迷宮的另一端。

solvemaze 常式如下：

```
function solvemaze(var maze: mazearray): boolean;
{ Attempt to solve maze, return TRUE if solved, else FALSE }

var
  solved: boolean;
  row, col: integer;
  tried: array [0..MAZEROWS, 0..MAZECOLS] of boolean;

{-----}
{ Module to be inserted here: }
{      try      }
{-----}

begin { solvemaze }
  for row := 0 to MAZEROWS do
    for col := 0 to MAZECOLS do
      tried[row, col] := FALSE;
  solved := FALSE;
  col := 0;
  row := 1;
  while not solved and (row < MAZEROWS) do begin
    solved := try(row, col, RIGHT);
    row := row + 1
  end;
  col := MAZECOLS;
  row := 1;
  while not solved and (row < MAZEROWS) do begin
    solved := try(row, col, LEFT);
    row := row + 1
  end;
  row := 0;
  col := 1;
  while not solved and (col < MAZECOLS) do begin
    solved := try(row, col, DOWN);
    col := col + 1
  end;
  row := MAZEROWS;
  col := 1;
  while not solved and (col < MAZECOLS) do begin
    solved := try(row, col, UP);
    col := col + 1
  end;
  solvemaze := solved
end;
```

solvemaze 所做的第一件事，是令布林 (Boolean) 陣列 **tried** 中的元素初值是 0 。這個陣列就像是迷宮地圖一樣，當迷宮中的每一格被尋求解答的程序訪問過後，它在 **tried** 中的相對元素，就變成 **TRUE** 了。

然後，**solvemaze** 尋求迷宮的邊界，企圖找出一個出口。它先往左邊找、再試右邊，上面、下面。(對所有經過 **cre** -

• 2 陰極射線管技術 •

atemaze 造出的迷宮，出口會在上端找到。）布耳變數 **solved** 被用來當做解答是否被找出的旗標，如果這迷宮無解，**solve-maze** 所含之值為 **FALSE**；否則是 **TRUE**。

布耳函數 **try** 之目的，在針對某一定點，某一定方向，偵查出是否有一條路徑通到邊界。例如，呼叫

try (row , col , up) ;

如果在第 **row** 行、第 **col** 列，往上的方向可以找到通往邊界的路徑的話，這個所呼叫的結果是 **TRUE**；否則就是 **FALSE**。就像在 **buildpath** 中一樣，**try** 也採用了遞迴（**recursion**）的方法來完成尋找的工作。

try 的虛擬代碼可表示如下：

```
begin
  if the specified square is a WALL
    no solution from this point
  else
    set tried flag for this square to TRUE
    consider neighboring square in specified direction
    if neighboring square is a WALL or has previously been examined
      no solution (at least not in specified direction)
    else
      display move to neighboring square on CRT
      if neighboring square is on boundary
        we've found a solution!
      else
        look for solution from neighboring point in each direction
        if no solution found
          erase previously displayed move
        end-else
      end-else
    end-else
  end-else
end
```

這個遞迴產生的主要關鍵，在於 **try** 一直由鄰近的點做各方向的尋找。寫成 **PASCAL** 如下：

```

function try(row, col: integer; dir: direction): boolean;
{ Attempt maze solution from point in given direction }

var
    ok: boolean;

{-----}
{ Modules to be inserted here: }
{           showmove           }
{           erasemove          }
{-----}

begin { try }
    ok := (maze[row, col] = PATH);
    if ok then begin
        tried[row, col] := TRUE;
        case dir of
            UP:
                row := row - 1;
            DOWN:
                row := row + 1;
            LEFT:
                col := col - 1;
            RIGHT:
                col := col + 1
        end;
        ok := (maze[row, col] = PATH) and not tried[row, col];
        if ok then begin
            showmove(row, col, dir);
            ok := (row <= 0) or (row >= MAZEROWS) or
                (col <= 0) or (col >= MAZECOLS);
            if not ok then
                ok := try(row, col, LEFT);
            if not ok then
                ok := try(row, col, DOWN);
            if not ok then
                ok := try(row, col, RIGHT);
            if not ok then
                ok := try(row, col, UP);
            if not ok then { no solution from this point }
                erasemove(row, col)
        end
    end;
    try := ok
end;

```

這一連串的解答過程，都將出現在螢幕上。因為每當有了移動的時候，**try** 會呼叫 **showmove**：

```

procedure showmove(row, col: integer; dir: direction);
{ Show move on CRT }

begin { showmove }
    case dir of
        UP, DOWN:
            dispsquare('!', row, col);
        RIGHT, LEFT:
            dispsquare('-', row, col)
    end
end;

```

• 2 陰極射線管技術 •

如果鄰近點再也找不出路來，則利用 **erasemove** 來取消剛走過的路：

```
procedure erasemove(row, col: integer);  
( Erase move on CRT )  
  
begin ( erasemove )  
    dispsquare(' ', row, col)  
end;
```

反復地呼叫 **showmove** 和 **erasemove**，可以使得當找到解答時，螢幕留下正確的途徑。

看 **theseus** 在螢幕上建立和解答迷宮十分有趣，而它所用到的工具，在許多比較嚴肅性的程式中，更具有價值。

建議 (Suggestion)

當你以比較嚴格的眼光來看一程式時，你絕對不會沒有收穫，**theseus** 也不例外。你也許想到利用第一章所學到的準則、來衡量這個程式。

看起來，**theseus** 只能應用在具有陰極射線管的交作環境之中，不具有易傳性 (non - portable)，它如修正以適應其他狀況呢？

如果你有一台列表機 (printer)，印出它最後找出的結果。除了字元之外，你也可以加上你的系統之中所擁有的繪圖能力，以美化迷宮。

你也許想試試看不同的演算法 (algorithm)，來建立和解答迷宮，你可以朝著改進效率的方向去做，或是只是要改變迷宮成不同的情況。無論是那一種，如果不使用遞迴性，這個演算會成為什麼樣子？在效率和清晰性上，它與具有遞迴性常式的演算法相比較又如何呢？

theseus 本身既建迷宮、又解迷宮，使用者很快就感到厭煩。你可以寫一個程式，讓使用者自己來解迷宮。同時，你也可以加入希臘神話故事，讓人身牛頭怪獸待在迷宮中來捕捉，並吃掉使用者。你並且可以嚐試各種不同的怪獸戰略。

推薦閱讀 (Recommended Reading)

由 APPLE PASCAL 所提供，在 APPLE 上能使用的 **crt** 軟體，稍有不同，而且所受到的限制也較多。但它的基本技術仍廣泛地被採用，並且出現在許多公用 PASCAL 裡。

本章中 **theseus** 所用的演算法，原出於 B. Kernighan 和 P.J. Plauger (McGraw - Hill , 1978) 所著的 **The Elements of Programming style** 的第四章。

3

交互輸入

(*Interactive Input*)

你曾有過犯了小錯而破壞了程式的經驗嗎？或是程式的反應令你迷惑嗎？當你鍵入輸入資料時，發生錯誤而導致輸入失敗，這又如何去更正呢？如你曾發生這些問題，表示你曾被一個「怠惰」的常式犧牲了；這個怠惰的輸入常式，往往使得程式使用困難、充滿挫折感、導致喪失一些有用的資料、浪費時間。

良好的交互輸入常式，使程式有信賴感，使用方便。最重要的法則是，其自使用者那接受資料時，要先核對所有輸入資料。

雖然要偵測出所有的錯誤，是不太可能的，但小心查核還是能解決一般性的問題，最佳之策是，偵錯容易，更正簡便。

本章的目的，在於發展和示範一個友善的交互式輸入工具，就像第二章的CRT程序，對於全書的程式，都非常有用。

標準的PASCAL本文輸入 (*Standard Pascal Text Input*)

標準PASCAL提供 **read** 和 **readln** 兩本文輸入的內建常式。這個原先就已設定好的常式（定義同 Jensen 和 wirth），

在整批環境 (batch environments) 中使用：由卡片、磁帶檔、磁碟檔等讀進資料。交作輸入時，使用 **read** 和 **readln**，會遭遇下列若干困難：

- 標準 (Jensen and wirth) PASCAL 要求，在程式開始執行之前，「預先——提取」輸入字元；此亦即，在程式要求之前即先鍵入第一個字元，這也是標準 PASCAL 的一般需求，(a) 在一檔案內 **reset**，將檔案的緩衝變數，與其檔案的第一個元素定初值。(b) 預定義檔案 **input** 在程式開始之際，蘊含 **reset**。
- **read** 和 **readln** 並不將 **end-of-file** 和 **end-of-line** 狀況，直接回轉；而 **eof** 和 **eoln** 被用來偵測這種情況。雖然，自檔案讀進字元，**eof** 和 **eoln** 兩函數有值，對於由鍵盤輸入，就並不很重要了。（這點是值得爭論的。尤其一個字鍵，不像一個磁碟檔可以產生一個不限制字元個數檔案；它無所謂起點或終點。）
- 當 **read** 和 **readln** 發生錯誤，PASCAL 無法使其復原，也無法將錯誤的字元抹除，使用者輸入錯誤，會導致程式變形或顯得奇怪。

總之，**read** 和 **readln** 並非適合大多數、基本的交錯輸入。這些不當的例子，常見諸於許多交作系統。例如，UCSD 和 APPLE PASCAL 定義一個新的本文檔 **interactive**，它不需要「預先——提取」第一個輸入字元，除非第一次用到 **read** 或 **readln**（接達緩衝變數 **input**），ISO 標準說明蘊含 **input** 的重置，並不一定發生。

鍵盤輸入 (Keyboard Input)

我們並不嘗試使 **read** 和 **readln** 工作敏感，而使得 **get key**

• 3 交互輸入 •

工作敏感。大部份的邏輯研究，皆設計 **getkey** 提供簡單的「基本的」作用：回復一個字元，來對應使用人鍵入字元，並不處理 end-of-file 或 end-of-line 字元；當使用人鍵入字元，**getkey** 即回復這些字元，給呼叫常式。

getkey 對於輸入字元，並無「回音」(echo)，也是值得注意的，**getkey** 僅取得字元，並不將字元在螢幕上排印。所以，**getkey** 在用取不排印的控制字元時，就非常簡單，用於 password 輸入益顯方便。

getkey 另外尚有二個特性。第一，它要求輸入字元是有效字元集之一，這是我們的第一個防止錯誤的要點，輸入字元如果不是有效字元，則該字元不被承認，並發出嗶嗶的聲音。

第二個特性是，用軟體的方法鎖住 shift 鍵：**getkey** 把輸入的小寫字母，轉換成大寫字母；由於它有這個特性程式限用大寫，或是大、小寫並用時，使得程式邏輯變得複雜時。**getkey** 表列如下：

```
function getkey(var ch: char; valid: charset; shiftlock: boolean): char;
{ Get valid key typed at keyboard, no echo }

var
  ok: boolean;
begin { getkey }
  repeat
    read(keyboard, ch);
    if eoln(keyboard) then
      ch := chr(13);
    if shiftlock and (ch in ['a'..'z']) then
      ch := chr(ord(ch) - 32);
    ok := ch in valid;
    if not ok then
      crl(HEEP);
  until ok;
  getkey := ch
end;
```

getkey 並非正統，却是常被使用的技巧：輸入字元以序數結果，和變數參數 **Var ch** 兩者，回復呼叫者，但前者可回

常式，可以立即測試剛鍵入的值，並儲存之，以便將來使用。

輸入參數 **valid** 指示 **getkey** 的輸入字元是合法的，並且 **getkey** 尚假定 **charset** 型態定義如下：

```
<type>
  charset = set of char;
```

此造成一個易傳性的問題：集合內可容下多少元素？爲了要使 **set of char** 合法化，一個集合可以擁有元素的個數與其不同字元數相同。APPLE PASCAL 可以有 512 個元素，因爲，APPLE PASCAL 的底線（underlying）字元集定義 256 個不同的字元。其他系統有更多的限制。（參看附錄 A）

getkey 的軟體鎖 shift 鍵，由布耳輸入參數 **shiftlock** 控制開關。該代碼段是：

```
if shiftlock and (ch in ['a'..'z']) then
  ch := chr(ord(ch) - 32);
```

如果 **shiftlock** 是 **TRUE**，則把小寫字母，轉換成大寫字母。數字 32 是 ASCII 字元集的大、小寫的差；本書將電子計算機的底線（underlying）字元當作 ASCII。

爲了完成沒有回音的輸入，**getkey** 利用 APPLE PASCAL（UCSD PASCAL 同）的特性：

```
read(keyboard, ch)
```

自鍵盤取得字元，而且不在螢幕顯示；如按下 < **RETURN** > 則函數 **coln (keyboard)** 回復 **TRUE**。

getkey 可由不同的硬體修改。例如，鍵盤並無 < ~ > 字元，**getkey** 則重組一個 2 一字元序 < **ESC** > < **6** >，來產生 < ~ >（輸入 < **ESC** > 字元，使用人可以鍵入 2 個 < **ESC** >），很顯然的，本方法允許使用人輸入任何鍵盤上沒有的 ASCII 字元。另外一種可能是，修正 **getkey** 以便處理一個

鍵盤的「特別功能」鍵，只要按一個鍵，就代表一序列字元。

有興趣的讀者，可能希望使用 `getkey` 來修飾前章的 `the-sens`。

字串輸入 (String Input)

現在，我們已能自鍵盤取得一個字元，下步驟是設計接受字串。字串對於 UCSD 而言，是一個內建資料型態 (`built-in data type`)，對於 APPLE PASCAL 而言亦然。大部份的 PASCAL 皆允許，程式設計師定義一個不定長度的字元，如你的版本，沒有這種內建資料型態，則你必須自行實行之。(見附錄 A)

`getstring` 是一個處理螢幕的源常式，它可在螢幕上任意點，接受輸入資訊。假如，一程式利用 `getstring`，輸入一個人名，於 CRT 上顯示

```
Subject's Name?:_____
```

底線字元指示輸入字元的最大個數，若超過長度，則多餘的字元，視同無效，並發出「嗶」的聲音。CRT 的游標落在第一個底線上，當使用人鍵入字串，它們即取代底線而成：

```
Subject's Name?:Thomas Je_____
```

鍵錯字元，可用 `<BACKSPACE>` (ASCII 8) 或 `<DELETE>` (ASCII 27) 鍵更正之。每一個 `<BACKSPACE>` 或 `<DELETE>` 抹除最後一個字元。如擬抹除更多的錯誤字元，就無法讓電子計算機或終端機發出「嗶」聲，前例中如連續按五次 `<BACKSPACE>`，則成：

```
Subject's Name?: Thom_____
```

另加一個編輯命令是 **< CONTROL - X >** (ASCII 24)
，鍵入這個命令，抹除整行，在更正錯誤後，如前繼續鍵：

```
Subject's Name?: Thomas Jefferson_____
```

輸入完成即按 **< RETURN >** (ASCII 13) 鍵。

資料登錄 (entry) 規則，對於讀者是非常熟悉了；在此所予的與 APPLE PASCAL 作業系統相容。使用人不需要去記憶不同的資料登錄方式。

getstring 將輸入的字串，轉換成所希望的資料型態，此外尚有其他好處，使用人不需要去記憶那些鍵入資訊的方法。

getstring 利用 **getkey** 去取得輸入字元，每輸入一字元，即行核對；只有合法的字元，可以連續鍵入。例如，使用人想要輸入一整數，**getstring** 可以規定，只接受數位、和 **< + >**、**< - >** 字元，拒絕其他字元。因為 **getstring** 使用 **getkey**，而 **getkey** 的軟體鎖 **shift** 可由 **getstring** 呼叫使用之。

getstring 有時取得缺設 (default) 答案。當使用人以 **< RETURN >** 鍵，取代登錄資料，就採缺設答案。明智的使用缺設，可以減少鍵入資料的時間。例如，

```
Loan Principal ($): $192.87_____
```

這時，192.87 即缺設值，使用人按 **< RETURN >** 時，程式即認為，使用人擬採缺設值 192.87 (注意，缺設可能在部份或全部的底線上，顯示登錄欄)。如果使用人，鍵入其他答案，缺設答案即不再出現，底線再度顯示出來，如使用人循序抹除所有鍵入的字元，則缺設答案即再度的顯現。

交互的輸入總是由程式在前導位置，給一個提示字元，告

• 3 交互輸入 •

訴使用人員，程式期待使用人鍵入一些資料，而且要使用人知道，該鍵入那一種資料。比如，提示 **DO you want to continue ? (Y/N)**：使用人只要鍵 **<Y>**（是之意），或鍵 **<N>**（不是之意）。或是提示 **Enter today's data (mm/dd/yy)**：使用人必須鍵入所示的格式之資料，如 **12/25/84**。

自使用人那取得資訊，普通有二種方法：第一種是「問題和回答」的方法，程式顯示一個提示（問問題），使用人回答所需要的資訊（答問題）。第二種是「填滿形式」法，使用人必須鍵入大量相對資料。在第二法中，所有的提示字元在 **CRT** 上顯示，而使用人依次回答每一問題，類似「填滿被排印的形式」。

getstring 並不提供字元，它只接受答案，提示必須由程式的其他部份提供之，所以 **getstring** 仍然適用於上式兩種方法。

getstring 的虛擬代碼（pseudo-code）如下：

```
begin
  set input string to null
  display underlines to show input field width
  display default, if any
  while input key isn't a <RETURN>
    if input key is a <BACKSPACE> or <DELETE>
      delete last character from input string
      erase last character on screen
    else if input key is a <control-X>
      erase entire line
      set input string to null
    else if input key is a valid character
      display it on screen
      add it to input string
    else
      signal user error
  end-while
  if input string is null
    set input string = default answer
  erase leftover underlines
end
```

可根據這點，直接寫 `getstring` 常式，只需修改 虛擬代碼 的不同的、不合法狀況（鍵入太多字元；希望抹除的字元，遠較已鍵入的多等等），和抹除或依據需要重顯示缺設答案。

`getstring` 完整表列如下：

```
procedure getstring(var inpstr: string; maxlen, col, row: integer;
                    default:string; valid:charset; shiftlock:boolean);
{ Get an input string from the user }

const
  BS = 8;           { ASCII backspace }
  CAN = 24;         { ASCII cancel }
  CR = 13;          { ASCII carriage return }
  DEL = 127;        { ASCII delete }
  FLDCHR = '_';     { Input field marker }

var
  ch: char;
  okset: charset;
  i: integer;
begin { getstring }
  inpstr := '';
  okset := valid + [chr(BS), chr(CR), chr(CAN), chr(DEL)];
  gotoxy(col, row);
  for i := 1 to maxlen do
    write(FLDCHR);
  if length(default) > 0 then
    posstr(default, col, row);
  gotoxy(col, row);
  while getkey(ch, okset, shiftlock) <> chr(CR) do begin
    if (ch in [chr(BS), chr(DEL)]) and (length(inpstr) > 0) then begin
      crt(LEFT);
      write(FLDCHR);
      crt(LEFT);
      chopchar(inpstr);
      if (length(inpstr) = 0) and (length(default) > 0) then begin
        write(default);
        gotoxy(col, row)
      end
    end
    else if (ch = chr(CAN)) and (length(inpstr) > 0) then begin
      gotoxy(col, row);
      for i := 1 to length(inpstr) do
        write(FLDCHR);
      gotoxy(col, row);
      inpstr := '';
      if length(default) > 0 then begin
        write(default);
        gotoxy(col, row)
      end
    end
    else if (ch in valid) and (length(inpstr) < maxlen) then begin
      if (length(inpstr) = 0) and (length(default) > 0) then begin
        for i := 1 to length(default) do
          write(FLDCHR);
        gotoxy(col, row)
      end;
      addchar(inpstr, ch, MAXSTR);
      write(ch)
    end
  end
```


• 3 交互輸入 •

```
end  
else ( Illegal character typed )  
  crt(BEEP)  
end;  
if length(inpstr) = 0 then begin  
  inpstr := default;  
  gotoxy(col + length(inpstr), row)  
end;  
for i := length(inpstr) + 1 to maxlen do  
  write(' ')  
end;
```

getstring 的輸入參數是：

- | | |
|-------------------|---------------------------------------------------------|
| maxlen | 輸入字串的最大長度 |
| col | CRT 的輸入行 |
| row | CRT 的輸入列數 |
| default | 必要時使用缺設答案，如果不需要缺設答案，本參數即設定為空字串（ null string ）。 |
| valid | 字串中可用的有效字元 |
| shift lock | 布耳值。如要轉換小寫字元，為大寫字元，則其值為 TRUE ，否則為 FALSE 。 |

輸入字元由變數 **inpstr** 返回。

更複雜的 **getstring** 可(1)在輸入字串中，不需要抹除再更正，而能夠隨意插入或刪除。(2)可以登錄更長的字串，甚至於超過 CRT 的邊緣。(3)提供鍵入不能排印的字元。(4)提供一個「求助鍵」，當使用人對於要鍵入字元有所迷惑，即可利用鍵，取得更多的資訊。程式加強這些特性並不具挑戰性，其困難在於，「增加這些特性，要使用人易於了解，記住這些特性的使用之」。

getstring 使用一個 APPLE PASCAL 的內建函數，却非標準 PASCAL 的內建函數：**length** 接受一個字串變數或常數作為引數，並以該數（字元的個數）返回。

字串操作常式 (String Manipulation Routines)

posstr 程序在 CRT 上所予的一位置，放置一字串如下：

```
procedure posstr(s: string; col, row: integer);
{ Put positioned string on CRT }

begin { posstr }
  gotoxy(col, row);
  write(s)
end;
```

addchar 程序將字串的最後，附加一字元：

```
procedure addchar(var s: string; ch: char; max: integer);
{ Add character to end of string }

begin { addchar }
  if length(s) < max then begin
    { $r- }
    s[0] := succ(s[0]);
    { $r+ }
    s[length(s)] := ch
  end
end;
```

本常式不具易傳性，在 APPLE 和 UCSD PASCAL 中，字串都是字元的列陣。 $s[i]$ 是字串中第 i 個字元， $s[0]$ 數值是該字串長度，如不查核字串範圍，就無從得到 $s[0]$ 之值，而查核由 $\{r-\}$ 實施。在 $s[0]$ 被接達 (access) 之後，範圍就與 $\{r+\}$ 一同回復。 $\{r+\}$ 和 $\{r-\}$ 是編譯名錄 (compiler directives)。

addchar 首先要保證有足夠的空間，去增加一些字元，使其最長可達 **max**。通常 **max** 設定為 **MAXSTR**。全盤變數 **MAXSTR** 被定義成字元在字串中最大的個數，通常定為 80。若有空間可附加字元， $s[0]$ 即加 1；亦即其長度加 1。

其他系統、版本的 PASCAL 中，可能有不同的 **addchar** 程序，也可能是內建程序。

第三個字串操縱常式是 **chopchar**，它自字串中刪除一個字元：

```
procedure chopchar(var s: string);  
( Delete character from end of string )  
  
begin ( chopchar )  
  if length(s) > 0 then  
    ($r-)  
    s[0] := pred(s[0])  
    ($r+)  
end;
```

本常式僅檢查字串中是否至少有一字元，是則字元的個數減 1。如同 **addchar**，在本版的 **chopchar** 中僅於 APPLE 和 PASCAL 上應用。

gaslog 常式

爲了發展交作輸入工具，設計 **gaslog** 來取得汽車使用汽油資料。本程式要儲存大量的資訊於磁碟，並於需要時排印報表，該資料檔是一系列的記錄所構成的，每一記錄有下列資訊：

- 汽油購買量
- 品牌 (Brand)
- 購油時里程 (odometer)
- 單價成本
- 購買的加侖數量
- 總消費

使用記錄的資料結構，來處理這些資料儲存，是 PASCAL 的特性。

使用人可利用 **gaslog** 增加檔案的記錄，或排印報表 以供參考。

當 **gaslog** 運轉，在 CRT 螢幕上方即顯示 **Gasoline Usage Log** 在螢幕下方顯示 **Initializing data file**

如果磁碟上尚無該檔，就建立一個檔，則在螢幕上顯示：

Creating data file

增加檔案之後，**gaslog** 即顯示資料登錄形式，如圖 3-1。
在螢幕底端出現問題 **Add more records ? (Y/N)**：

如要再增加記錄，鍵入 **<Y>**，不再增加，即回答 **<N>**。

如使用人希望加上更多資料，程式需要購買油料數據，每回答一個項目，就提示下一項，它採「格式填滿」的方法，此處對於各種資訊指導如下：

Date： 只有數位和斜線 **</>** 字元合文法，缺設 **date** 取自前一記錄，其形式是 **mm/dd/yy**；月份 (**mm**) 範圍是 1 到 12，日期 (**dd**) 是 1 到 31，年份是 0 到 99。如果程式偵到任何不合文法的資料，即顯示 **Please enter date in the form mm/dd/yy**，以供使用人重新鍵入合法的資料。

<div style="text-align: center;">Gasoline Usage Log 39 Records Used Out of 200</div> <div>Record Number: Date: Brand: Odometer Reading: Price (cents/gallon): Amount (gallons): Total Cost:</div> <div style="text-align: right;">Add more records?(Y/N): _</div>

• 3 交作輸入 •

汽油使用資料是按照時間序列登錄，違規即得下列警告訊號：

```
WARNING - Before last record's date
          Press <RETURN> to continue
```

使用人按< RETURN >，程式即提示下一個資料項目。通常，除非是錯誤太明顯，程式最好採用「使用人知道最好」的態度。

為增快速度，使用人只要鍵入月份和日期，年份採缺設。例如，缺設日期 10/13/84，使用人只鍵入 10/26，則登錄成 10/26/84。

Brand 資料登錄限制 10 個字元；所有可印的字元，均轉換成大寫、缺設值與上一記錄相同。

Odometer Reading 僅允許數位和< . >，< + >，和< - >字元，如鍵入數字小於 0 或大於 999999.9，則顯示

```
Please enter a number between
          0.0 and 999999.9
```

並再度要求鍵入里程，若鍵入資料小於或等於上一個記錄，則警告使用人：

```
WARNING - mileage <= prior mileage
          Press <RETURN> to continue
```

鍵入< RETURN >，程式詢問下一個項目。

Price (cents / gallon)： 本項目登錄資料也是數字資料、法則與上個項目一樣，成本是每加侖多少分；例如，成本是 \$ 1.258，即鍵入 125.8，缺設值與上一記錄的燃料價格相同，而單位價格不得小於 0.1 分 / 加侖，或大於 \$ 3.00 / 加侖。

Amount (gallons)： 合法量介於 0.01 和 99.99 加侖之間，沒有缺設值。

TOTAL Cost： 合法值介乎 \$ 0.01 和 \$ 3.00 之間，缺設

值根據燃油成本和購買量求出的，若汽油每加侖 123.6 分，12.3 加侖的購油量、缺設值是 \$ 15.20 (1.236×12.3 ，小數點取 2 位，其餘捨之)

使用人鍵入的數目，與估計的總成本相差 2 %，即發出警告。

以上六項全部鍵入，程式問 **Any changes ? (Y/N)**：鍵入 < Y > 則上述六項值全部重新鍵入，處理過程中，可以鍵 < RETURN > 表該項採其次的答案，只要更正那些錯誤的項目即可。

更改完畢之後，程式再度詢問：**Any changes ? (Y/N)**：反覆作此循環，直到更正到令使用人滿意為止。則程式顯示：**Ok to add record ? (Y/N)**：若流程記錄全錯、或鍵入錯誤資料，就回答 < N >，則該記錄就不加到檔案上去，否則使用人該回答 < Y >。

此後程式詢問 **Add more records ? (Y/N)**：如回答 < Y >，程式需要一新的記錄。

新增記錄完成之後，程式詢問 **Print gas usage report ? (Y/N)**：如果想要一份用油報告，就回答 < Y >，程式即要求使用人，將今天日期鍵入，以便在每頁的上方排印今天日期。見圖 3-2。

使用人以是與否的答案，即能控制 **gaslog** 流程，這對於使用人而言，很容易操作，也很容易了解程式。比較複雜的程式，其控制流程也就複雜。如果想要刪除或更新資料檔案的記錄，就必須有一個接達這些選擇的方法，而是與否的簡單結構，即不允考慮。

一個最後設計的要點，對於使用人而言，很多資訊（問題、指令、警告、錯誤指示等等）皆在 **CRT** 的底端二行顯現。**gaslog** 並不使用這二行顯示，作為正常顯示，我們也利用有

• 3 交互輸入 •

聲信號（呼叫 `crt (BEEP)`）指示一些錯誤，提醒使用人注意，諸如在 `getkey` 接收一不合法的字元，或 `getstring` 收到更多的字元。

Gasoline Usage Report						09/15/82
						Page 1
Date	Brand	Odometer Reading	Fuel Cost (cents/gal)	Amount (gals)	Total Cost	MPG
08/02/82	Mobil	45062.1	132.9	9.50	12.60	----
08/05/82	Mobil	45310.9	130.8	9.70	12.70	25.6
08/10/82	Old Colony	45602.1	129.9	9.90	12.90	29.4
08/15/82	Exxon	45888.2	132.9	8.50	11.30	33.7
08/18/82	Mobil	46201.6	127.5	9.80	12.50	32.0
08/25/82	Mobil	46449.2	127.5	8.90	11.35	27.8
09/02/82	BP	46751.4	124.9	9.80	12.25	30.8
09/05/82	Lido	47002.5	123.6	8.40	10.40	29.9
09/12/82	Amoco	47298.4	126.9	7.90	10.00	37.5
09/15/82	Getty	47560.6	129.9	8.60	11.20	30.6

圖 3-2 用油報表

茲另舉一個敏感的使用人——原始的設計為例，通常使用人發生錯誤，程式就要求使用人做一些動作，儘可能的繼續處理之。如同第一章所討論的一樣，一貫性是程式的容易使用的重要特性。

`gaslog` 的主常式虛擬代碼設計如下：

```

begin
  initialize data file (create it if necessary)
  display number of records in file
  display capacity of file
  display data entry form
  repeat
    if there's room in the file for more records
      ask user if there are more records to add
    if there are more records to add and there's room in the file
      get gas purchase data from user
      ask if it's OK to add record to file
      if it's OK
        write record to file
        display new number of records
  until file is full or there are no more records to add
  if the user wants a gas report
    print a gas report
  close data file
end

```

改成PASCAL 則如下：

```
{ $s+ }
program gaslog;
{ Maintain gasoline usage records }

const
  BRANDSIZE = 10;      { Maximum length of gasoline brand }
  FIXSIZE = 12;        { Maximum digits in fixed number }
  MAXCRTCOL = 79;      { Maximum CRT column number }
  MAXCRTROW = 23;      { Maximum CRT row number }
  MAXRECS = 200;       { Maximum records in data file }
  MAXSTR = 80;         { Maximum length of string }

type
  crtcommand = (HOME, CLEAR, ERASEOL, ERASEOS, UP, DOWN, LEFT, RIGHT, BEEP);
  charset = set of char;
  date = packed record
    month: 0..12;
    day: 0..31;
    year: 0..100
  end;
  fixed = integer[FIXSIZE];
  gasrec = record
    fdate: date;
    brand: string[BRANDSIZE];
    mileage, fuelcost, gallons, totcost: fixed
  end;
  gasfile = file of gasrec;

var
  gf: gasfile;
  margin, nrecs: integer;

  lastrec, newrec: gasrec;
  s1, s2: string;
  filefull, more: boolean;

{-----}
{ Modules to be included here: }
{      crt      }
{      posstr   }
{      getkey   }
{      addchar  }
{      chopchar }
{      getstring }
{      eraseline }
{      center   }
{      getboolean }
{      ask      }
{      gchar    }
{      gnbchar  }
{      stof     }
{      ftoS     }
{      getfixed }
{      dtos     }
{      getdate  }
{      wait     }
{      remark   }
{      readgasrec }
```


• 3 交作輸入 •

```
{      writegasrec      }
{      initgasfile      }
{      dispgasform      }
{      getgasrec         }
{      gasreport         }
{-----}

begin ( gaslog )
  margin := (MAXCRTCOL - 25) div 2;
  crt(CLEAR);
  center('Gasoline Usage Log', 0);
  initgasfile(gf, nrecs, lastrec);
  ftos(MAXRECS, 0, 0, s2);
  s2 := concat(' records used out of ', s2);
  ftos(nrecs, 0, 0, s1);
  center(concat(s1, s2), 2);
  dispgasform;
  more := TRUE;
  repeat
    filefull := (nrecs >= MAXRECS);
    if not filefull then begin
      more := ask('Add more records?(Y/N):', MAXCRTROW - 1, TRUE, TRUE);
      eraseline(MAXCRTROW - 1)
    end;
    if more and not filefull then begin
      getgasrec(newrec, lastrec, nrecs);
      if ask('OK to add record?(Y/N):', MAXCRTROW - 1, TRUE, TRUE) then begin
        nrecs := nrecs + 1;
        writegasrec(gf, nrecs, newrec);
        ftos(nrecs, 0, 0, s1);
        center(concat(s1, s2), 2);
        lastrec := newrec
      end
    end
  until filefull or not more;
  if ask('Print gas usage report?(Y/N):', MAXCRTROW - 1, TRUE, FALSE) then
    gasreport(gf, nrecs);
  close(gf);
  crt(CLEAR)
end.
```

gaslog 程式很大，有足夠的理由要求APPLE PASCAL 編譯器採用「調換模式」，這由程式的第一行 { \$S+ } 完成之，如果沒有這個指引，在編輯階段，記憶體即不敷使用。本書並不對APPLE PASCAL的編譯工作，做太詳細的探討，只需略加注意，「調換」模式有助於記憶體的有效使用，其他版本的PASCAL未必具有此一特性。

資料結構 (Data Structure)

gaslog 尚需兩種尚未討論到的資料型態 (data type) ,
第一種是 **date** , 用於儲存日曆日期, **date** 定義如下:

```
<type>
  date = packed record
    month: 0..12;
    day: 0..31;
    year: 0..100
  end;
```

型態變數 **date** 由次範圍 (subrange) 組成, 而 **packed** 指示 PASCAL 編譯器用最小的儲存體、儲存資料型態, 在 APPLE PASCAL 以緊湊的方法來代表日期, 僅僅用到 2 個位元組, 這方法與 APPLE PASCAL 的作業系統 (operating system) 的儲存日期的方法相容。

第二種資料型態是

```
<type>
  fixed = integer[FIXSIZE];
```

而 **FIXSIZE** 定義同前:

```
<const>
  FIXSIZE = 12;
```

這個定義使用到長整數 (long integer), 它並非 APPLE 和 UCSD Pascal 提供的非一標準資料型態。APPLE PASCAL 的整數範圍是 - 32767 到 + 32767。長整數定義比較大的整數範圍, 例如:

```
<var>
  count: integer[8];
```

count 變數可掌握的整數是 8 位數, 其範圍是 - 99999999 到 99999999。長整數的最大限制是 36 位數。

現在已相當了解 **fixed** 資料型態, 在 **gaslog** 中, 即用

• 3 交作輸入 •

fixed 變數代表數字，Monetary 值就表示便士 (pennies) 的整數的數目；例如 \$ 123.87 即 12387 便士，燃料總數 (Fuel amounts) 加侖的百倍數，10.50 加侖，則 Fuel amount 即 1050，車輛里程表 (mileage) 是里程數的 10 倍，46123.5 哩就變成整數 461235，最後燃料成本 (cost of fuel) 則為每加侖單價 (分) 的 10 倍，如 \$ 1.258 / 加侖，該值變成整數就成為 1258。

這個方法叫「定點表示法」，於是我們稱為資料型態 **fixed**。使用這個方法，程式必須知道小數點位置在那裡。用油總量 1289 就是 12.89 加侖，以 1289 表示油料單價，則該值就變成每加侖 \$ 1.289。計算過程要格外小心。

fixed 資料型態最重要的用途，是控制資料的準確度，表示十億元時都不少掉一個便士，雖然使用 **gaslog** 不一定要這麼精確，我們依然將它做成內建工具，以供其他方面應用之，**gaslog** 亦較我們所需要的還要準確。

有許多版本 PASCAL 提供廣域精度 (extended-precision) 整數資料型態，如你的 PASCAL 不具備此，可以自己加上。(見附錄 A)

資料檔定義如下：

```
<type>
gasrec = record
  fdate: date;
  brand: string[BRANDSIZE];
  mileage, fuelcost, gallons, totcost: fixed
end;
gasfile = file of gasrec;
```

gasrec 被定義成：將一個人購買的所有項目累積到一個單一記錄型態，其以順序記錄的形式存在磁碟檔案中。

字串序連 (String Concatenation)

在 **gaslog** 中另有一常式 **concat**，它並非標準 **pascal** 的一部份，它可以把二個或更多的字串序連 (concatenate) 成一個字串，例如：

```
s1 := 'one, ';  
s2 := 'two, ';  
s3 := 'three.';  
s := concat(s1, s2, s3);  
write(s);
```

顯示字串 **one, two, three**

concat 接受任意個字串引數，並且以一個字串作為函數結果回轉之。在標準 **pascal** 中，可由程式設計師自行設計之，該設計的程序或函數，有一固定數的引數，轉回時是一個純量 (scalar) 結果。(詳見附錄 A)

布耳輸入 (Boolean Input)

使用人回答 **yes** 或 **no**，來控制 **gaslog** 流程，諸如：

Addmore record ? , print gas usage report ? , 等等。

這些普通應用到大部份的程式去，所以值得設計一個常式去處理。這些問題中，用到一函數 **ask**，當使用人回答 **yes**，它的結果是 **TRUE**，反之，其結果為 **FALSE**：

```
function ask(prompt: string; row: integer; defbool, defaulted: boolean): boolean;  
( Ask user yes-or-no question )  
  
begin ( ask )  
  center(prompt, row);  
  ask := getboolean((MAXCRTCOL+length(prompt)+1) div 2, row, defbool, defaulted)  
end;
```

• 3 交互輸入 •

ask 的輸入參數是：

prompt 被問的問題
row 問題所在列
defbool 缺設答案，yes 時是 TRUE，NO 時是 FALSE。
defaulted 可以有缺設回答時是 TRUE，否則 FALSE，如 FALSE，defbool 就無意義。

center 常式設用於，在 CRT 的中央部份顯示一個字串：

```
procedure center(s: string; row: integer);
( Center line on CRT )

begin ( center )
  eraseline(row);
  posstr(s, (MAXCRTCOL - length(s) + 1) div 2, row)
end;
```

爲了確定字串被 **center** 顯示，就先要用 **eraseline** 抹除之：

```
procedure eraseline(row: integer);
( Erase line on CRT )

begin ( eraseline )
  gotoxy(0, row);
  crt(ERASEOL)
end;
```

小心使用 yes 或 no，**getboolean** 接受使用人的回應。該常式在 CRT 上特定點上，接受使用人的回答（也接受缺設回答）：

```
function getboolean(col, row: integer; defbool, defaulted: boolean): boolean;
( Get yes-or-no from user )
```

```
var
  defstr, inpstr: string;
  ok, booboo: boolean;

begin { getboolean }
  if not defaulted then
    defstr := ''
  else if defbool then
    defstr := 'Y'
  else
    defstr := 'N';
  booboo := FALSE;
  repeat
    getstring(inpstr, 1, col, row, defstr, ['Y', 'N'], TRUE);
    ok := (length(inpstr) = 1);
    if not ok then begin
      booboo := TRUE;
      crt(BEEP);
      center('Please enter "Y" or "N"', MAXCRTROW)
    end
  until ok;
  if booboo then
    eraseline(MAXCRTROW);
  getboolean := (inpstr = 'Y')
end;
```

讀者可以根據本常式，推論 yes 或 no 的法則。

定點數值的輸入 (Fixed-Point Numeric Input)

getboolean 不過是一個呼叫 **getstring** 的例子, **getstring** 將一個特定的資料型態當作一字串，再根據需要轉換成所希望的資料型態，用虛擬代碼來表示資訊方法如下：

```
begin
  set up parameters to getstring:
    maximum length
    valid characters
    convert default to a string
  repeat
    get input string from user (using getstring)
    convert it from a string to desired data type
    check it for validity
    if it's valid
      convert it to standard form and display it (if desired)
    else
      signal user error, offer advice
  until valid input
end
```

• 3 交作輸入 •

這是一般性的戰略，描述並不詳盡，這些步驟中，有的非常簡單，而且無必要，有的複雜，把上述代碼，改寫成 PASCAL 的常式如下：

```
procedure getfixed(var f: fixed; col, row: integer; min, max, deffix: fixed;
                   ndigs: integer; defaulted: boolean);
{ Get fixed number from user }
var
  minstr, maxstr, defstr, s: string;
  maxlen, i: integer;
  booboo, good: boolean;
begin { getfixed }
  ftostr(min, 0, ndigs, minstr);
  ftostr(max, 0, ndigs, maxstr);
  if length(minstr) > length(maxstr) then
    maxlen := length(minstr)
  else
    maxlen := length(maxstr);
  if defaulted then
    ftostr(deffix, 0, ndigs, defstr)
  else
    defstr := '';
  booboo := FALSE;
  repeat
    getstring(s, maxlen, col, row, defstr, ['0'..'9', '+', '-', '.'], FALSE);
    good := (length(s) > 0);
    if good then begin
      i := 1;
      good := stof(s, i, ndigs, f)
    end;
    if good then
      good := (f >= min) and (f <= max);
    if good then begin
      ftostr(f, maxlen, ndigs, s);
      posstr(s, col, row)
    end
    else begin
      booboo := TRUE;
      crt(BEEP);
      center('Please enter a number between', MAXCRTROW - 1);
      center(concat(minstr, ' and ', maxstr), MAXCRTROW)
    end
  until good;
  if booboo then begin
    gotoxy(0, MAXCRTROW - 1);
    crt(ERASEOS)
  end
end;
```

getfixed 輸入參數是：

col , row CRT 上的位置（取得數目）。

min	取自使用人的最小值。
max	自使用人處取得最大的值。
deffix	缺設答案。
ndigs	在答案中十進位數值的個數（指小數點後）。
defaulted	缺設回答如所願則為 TRUE，否則為 FALSE。若為 FALSE， deffix 即無意義。

上述這些輸入數字，由參數 **f** 回轉。數字由 **ndigs** 來轉換成 **fixed** 的資料型態。如果 **ndigs** 是 3，而鍵入字串是「3.142」，**getfixed** 轉回的答案是 3142，如 **ndigs** 是 2，則回轉答案是 314。

getfixed 還有二個常數沒有被介紹過，第一是 **ftos**，把一定數轉換成一字串。**ftas** 有二個輸入參數 **width** 和 **ndigs**，這二參數來控制轉換格式，**ndigs** 位數在小數點後面出現，而其左方則補上空白，而其長度可到 **width** 個字元，它的演算複雜，其虛擬代碼大綱列舉如下（相當複雜）：

```
begin
  if number is negative
    set f to -f
    remember that f was negative
  set string to null
  set number of digits converted to 0
  repeat
    get units digit of f
    add digit character to string
    set f to (f div 10)
    increase number of digits converted by one
    if number of digits converted = ndigs
      add decimal point to string
  until (f = 0) and (digits converted > ndigs)
  if f was negative
    add minus sign to end of string
  while length(s) < width
    add space to end of s
  reverse string
end
```

ftos 的唯一技巧是，它建立字串是相反次序，並於回轉

呼叫常式之前換位。

```

procedure ftostr(f: fixed; width, ndigs: integer; var s: string);
( Convert fixed to string )

var
  negnum: boolean;
  f1: fixed;
  i, j, nc: integer;
  ch: char;

begin ( ftostr )
  negnum := (f < 0);
  if negnum then
    f := -f;
  s := '';
  nc := 0;
  repeat
    f1 := f div 10;
    addchar(s, chr(trunc(f - 10 * f1) + 48), MAXSTR);
    f := f1;
    nc := nc + 1;
    if nc = ndigs then
      addchar(s, '.', MAXSTR)
  until (f = 0) and (nc > ndigs);
  if negnum then
    addchar(s, '-', MAXSTR);
  while length(s) < width do
    addchar(s, ' ', MAXSTR);
  i := 1;
  j := length(s);
  while i < j do begin
    ch := s[i];
    s[i] := s[j];
    s[j] := ch;
    i := i + 1;
    j := j - 1
  end
end;

```

trunc 函數接受一長整數，做為一引數，並且回轉一個正常整數，標準 PASCAL 的客顧把它加到系統裡去，它接受一實數引數 (argument) 回復一整數。無論是那一種，如果引數值超過 MAXINT，就會產生溢位錯誤。

APPLE 和 UCSD PASCAL 尚有一特性，即它需要一長整數，而接受的是整數，它就自動轉換，在 **gaslog** 的主常

式可以看到例子。

第二個常式是 **stof**，把一字轉換成一定數。

```
ok := stof(s, i, ndigs, f);
```

掃描 S 字串，自第 i 個字元起，以定數 f 回轉之。**ndigs** 是指小數點的位置之後的位置，不斷的掃描，直到字串完畢，或到不能看見的字元為止。此時，變數 i 回復字串轉換終止的位置。

stof 回轉一個布耳值，做為函數結果，數字是否在字串中順利的翻譯到 **fixed** 變數，如果工作順利，就回轉 **TRUE**，如果數字太大，超過 **FIXSIZE** 位數，它就回轉 **FALSE**。

stof 保留在變數 **nsig** 中的有效位數，例如一個 14 位數的數 00000000.000042，有效位數只有 2 位。如果有效位數超過 **FIXSIZE** 的有效位數，超過的位數就忽略掉。

小數點後面的位數就留在 **nafter** 變數；若果在小數點後面看到的位數，大於 **ndigs**，它們同樣的把超過的部份略過不要，否則就無意義；如果少於 **ndigs**，則輸出的 **fixed** 數變為 10 的乘方。

stof 函數如下：

```
function stof(var s: string; var i: integer; ndigs: integer; var f: fixed):  
boolean;  
( Convert string to fixed )  
  
var  
    negnum: boolean;  
    nsig, nafter: integer;  
    c: char;  
  
begin ( stof )  
    f := 0;  
    nsig := 0;  
    nafter := 0;  
    negnum := FALSE;  
    if gnbchar(s, i, c) in ['+', '-'] then begin  
        negnum := (c = '-');  
        i := i + 1  
    end;  
    while gnbchar(s, i, c) = '0' do  
        i := i + 1;  
    while gnbchar(s, i, c) in ['0'..'9'] do begin
```

```

    if nsig < FIXSIZE then
        f := 10 * f + ord(c) - 48;
        i := i + 1;
        nsig := nsig + 1
    end;
    if c = '.' then begin
        i := i + 1;
        if nsig = 0 then
            while gnbchar(s, i, c) = '0' do begin
                nafter := nafter + 1;
                i := i + 1
            end;
            while gnbchar(s, i, c) in ['0'..'9'] do begin
                if (nsig < FIXSIZE) and (nafter < ndigs) then
                    f := 10 * f + ord(c) - 48;
                i := i + 1;
                nsig := nsig + 1;
                nafter := nafter + 1
            end
        end;
        if (nafter < ndigs) and (f <> 0) then
            while (nsig < FIXSIZE) and (nafter < ndigs) do begin
                f := 10 * f;
                nsig := nsig + 1;
                nafter := nafter + 1
            end;
        if negnum then
            f := -f;
        stof := (nsig - nafter) <= (FIXSIZE - ndigs)
    end;
end;

```

stof 允許在一數加上空白，它掃描下一個非空白的字元，有興趣的讀者可考慮把它應用到其他狀況。

stof 把小數點後面 **ndigs** 的位置之數字捨棄，把它修訂，採取捨入非常困難，它有什麼好處和壞處。

接達字串字元 (Accessing String Characters)

gnbchar 在 **stof** 內掃描非空白字元：

```

function gnbchar(var s: string; var i: integer; var c: char): char;
{ Get next non-blank character from string }

begin
    while gchar(s, i, c) = ' ' do
        i := i + 1;
    gnbchar := c
end;

```

由字串的第 i 字元開始掃描，如第 i 個字元是一空白，就掃描下一字元，當掃描到結尾，它就計數那非空白字元。非空白字元（字串的結尾就是 $\langle \text{NIL} \rangle$ ，ASCII 0）以函數結果和變數 c 回轉。var 參數 i 回復第一個非空白的位置。

利用 **gchar** 常式自字串抽取字元：

```
function gchar(var s: string; i: integer; var c: char): char;
{ Get character from string }

begin { gchar }
  if (i < 1) or (i > length(s)) then
    c := chr(0)
  else
    c := s[i];
  gchar := c
end;
```

gchar 回轉字串 s 的第 i 個字元，並將二者作為函數結果和 var 參數 c 。如果 i 超過字串的限制之外面，常式就回轉 $\langle \text{NUL} \rangle$ 字元。

日期輸入 (Date Input)

下一步驟寫 **getdate**：

```
procedure getdate(var inpdate: date; col, row: integer; defdate: date;
                  defaulted: boolean);
{ Get date from user }

var
  defstr, inpstr: string;
  booboo, good: boolean;

{-----}
{ Modules to be included here: }
{      stod      }
{-----}

begin { getdate }
  booboo := FALSE;
  if defaulted then
    dtos(defdate, defstr)
  else
    defstr := '';
  repeat
    getstring(inpstr, 8, col, row, defstr, ['0'..'9', '/'], FALSE);
    stod(inpstr, inpdate);
    with inpdate do begin
      if (year = 100) and defaulted then
        year := defdate.year;
```

• 3 交互輸入 •

```
    good := (month in [1..12]) and (day in [1..31]) and (year in [0..99])
  end;
  if good then begin
    dtos(inpdate, inpstr);
    posstr(inpstr, col, row)
  end
  else begin
    booboo := TRUE;
    crt(BEEP);
    center('Please enter a date in the form mm/dd/yy', MAXCROW)
  end
until good;
if booboo then
  eraseline(MAXCROW)
end;
```

此外，本常式跟在特定型態的輸入，**getdate** 利用 **stod** 程序，將字串轉換成日期：

```
procedure stod(var s: string; var d: date);
{ Convert string to date }

var
  i: integer;
  f: fixed;
  c: char;
begin { stod }
  with d do begin
    month := 0;
    day := 0;
    year := 100;
    i := 1;
    if stof(s, i, 0, f) then
      if (f >= 1) and (f <= 12) then
        month := trunc(f);
    if gnbchar(s, i, c) = '/' then
      i := i + 1;
    if stof(s, i, 0, f) then
      if (f >= 1) and (f <= 31) then
        day := trunc(f);
    if gnbchar(s, i, c) = '/' then begin
      i := i + 1;
      if stof(s, i, 0, f) then
        if (f >= 0) and (f <= 99) then
          year := trunc(f)
    end
  end
end;
```

stod 利用 **stof** 轉換常式，加上 **trunc** 函數，來轉換日期字串的每一部份，成為整數。

最後，**dtos** 將 **date** 轉換成一個字串，它相當短、也簡單

:

```
procedure dtos(d: date; var s: string);
{ Convert date to string }

begin { dtos }
  s := '00/00/00';
```

```
with d do begin
  year := year mod 100;
  s[1] := chr(month div 10 + 48);
  s[2] := chr(month mod 10 + 48);
  s[4] := chr(day div 10 + 48);
  s[5] := chr(day mod 10 + 48);
  s[7] := chr(year div 10 + 48);
  s[8] := chr(year mod 10 + 48);
end;
end;
```

尚有一些事物值得注意。第一，如果比較喜歡其他的格式（例如 15 — step — 84，而非 9 / 15 / 84），你可改變這三個常式。第二，**stod** 可以捕捉許多「不合文法」的日期，而無法指出全部錯誤。例如，23 / 45 / 67 是很容易發覺，因為第二項 45 超過範圍，然而 4 / 31 / 51 就視同正確了。

資料檔案初域 (Data File Initialization)

現在已建好數字和日期的輸入常式，我們可回過頭來寫 **gaslog**，第一個常式是 **initgasfile**，它打開既有的汽油資料檔或建立一個新的汽油資料檔。

第一次運轉 **gaslog** 時並沒有資料檔，就必須建立一個檔，應採取下列一、二個戰略。(1)建立一個空檔。(2)建立一個檔，該檔有最大量的記錄，並且註記它是一個空記錄。第一種情況，新記錄就加到檔案的最後，則檔案稍大，也占較多的空間。第二種情形，新記錄就取代空記錄，檔案占相同的位置。

雖然我們不需要太多技巧去知道資料如何存到磁碟，但是 **APPLE** 和 **UCSD PASCAL** 必須有一磁碟檔，以占據連續的磁碟塊，這也就是說，檔案 A 在檔案 B 之前面，它們中間沒有任何塊被佔據，檔案 A 無法加大，所以我們大力推薦使用第二種方法，則不僅可以放心，也絕無法在我們的資料檔之後，立即放置一個檔，也能防止它不斷擴張變大。

如果主作業系統 (host operation system)，允許檔案

占據非連續塊，就無需擔心這些問題了。所以值得在固定大小檔案後面採用這種戰略，尤其磁碟容量有限制的時候。如果磁碟包括很多其他檔案時，增加一個汽油資料檔，會超過整個磁碟的容量。

增長檔案方法是由一分時系統指示，使用人付磁碟儲存的全部金錢，在資料檔中浪費太多空間，去儲存許多空的記錄。

我們使用固定大小檔案方法，設計 **gaslog** 程式，當最初檔案被建立時，寫 **MAXRECS** 空記錄 **MAXRECS** 在前面被定義的常數，取之為 200)，年份 100 的記錄，視為一空記錄。年份 100 是個不合文法的值，無法由使用人鍵入的值，在 **stod** 和 **getdate** 都會檢查出的錯誤。

如汽油資料檔存在，**initgasfile** 利用二分搜尋去找檔案裡的第一個空記錄，要加記錄就在這點開始。該常式也讀最後鍵入的記錄；如果它存在，就用來做為下一記錄的缺設值。

```

procedure initgasfile(var gf:gasfile; var nrecs: integer; var lastrec: gasrec);
{ Initialize gas file, create if necessary }

const
  GASFILE = 'GASLOG.DATA';      { Name of data file }

var
  grecc: gasrec;
  lorec, hirec, i: integer;

{-----}
{ Module to be included here: }
{      findfile                  }
{-----}

begin { initgasfile }
  center('Initializing data file...', MAXCRTROW);
  if findfile(GASFILE) then begin
    reset(gf, GASFILE);
    lorec := 0;
    hirec := MAXRECS + 1;
    while hirec - lorec > 1 do begin
      i := (hirec + lorec) div 2;
      readgasrec(gf, i, grecc);
      if grecc.fdate.year = 100 then
        hirec := i
      else
        lorec := i
    end
  end
end

```

```

end;
if lorec > 0 then
  readgasrec(gf, lorec, lastrec);
nrecs := lorec
end
else begin
  center('Creating data file...', MAXCRTROW);
  with grec do begin
    with fdate do begin
      month := 0;
      day := 0;
      year := 100
    end;
    brand := '';
    mileage := 0;
    fuelcost := 0;
    gallons := 0;
    totcost := 0
  end;
  rewrite(gf, GASFILE);
  for i := 1 to MAXRECS do
    writegasrec(gf, i, grec);
  close(gf, LOCK);
  reset(gf, GASFILE);
  nrecs := 0;
  lastrec := grec
end;
eraseline(MAXCRTROW)
end;

```

本常式尚要忍受二個缺點。第一，它假定當它建立資料檔時有足夠的空間去容納資料檔。如果空間不敷使用，就有致命的錯誤發生，唯一可能改進的是，當建檔時，先關掉 I/O 的錯誤查核，當發生錯誤時，再向使用人報告錯誤。

第二個缺點是資料檔的位置，缺乏彈性；假設資料檔是在「缺設磁碟」，使用人必須能告訴程式，資料檔放在那裡。

initgasfile 利用常式 **findfile**，來決定資料檔存在與否；如資料檔在磁碟上 **findfile** 回轉 **TRUE**，否則回轉 **FALSE**。

```

function findfile(name: string): boolean;
( Look for file, return TRUE if found, else FALSE )

var
  found: boolean;
  f: file;

begin ( findfile )
  ($i-)
  reset(f, name);

```


• 3 交作輸入 •

```
found := (ioresult = 0);
{$i+}
if found then
  close(f);
findfile := found
end;
```

findfile 是非易傳性的，標準 PASCAL 無法用檔案名稱接達檔案，或決定該檔是否存在 APPLE 和 UCSD PASCAL，利用 **reset (f, name)**，去打開用字串名稱註明的檔，並可被檔案變數 **f** 接達 (**access**)。如果該檔不存在，而要 **reset**，就被認為 I/O 有誤，並且正常的終止程式執行，{ \$i - } 指引名錄 (**directive**) 允許在 I/O 發生錯誤時，繼續運轉；錯誤可由 **ioresult** 來註記，回轉一個非零的值。指引名錄 { \$i + } 恢復正常的 I/O 查核。

宣告部份

```
<var>
  f: file;
```

也是非易傳性的，APPLE 和 UCSD PASCAL 允許這個「非型態」檔案宣告，並允許直接操作一檔案的塊。可用一般的方法寫 **findfile**：它是不是一個本文檔、一個資料檔，或一個程式檔。

readgasrec 函數自檔案讀一個特定的記錄：

```
procedure readgasrec(var gf: gasfile; recno: integer; var grec: gasrec);
( Read gas record from file )

begin ( readgasrec )
  seek(gf, recno - 1);
  get(gf);
  grec := gf^
end;
```

在標準 PASCAL 中，檔案緩衝變數 **gf** 包括所要的記錄。

writegasrec 是相反的程序，寫一個記錄到檔案中：

```
procedure writegasrec(var gf: gasfile; recno: integer; var grec: gasrec);  
( Write gas record to file )  
  
begin { writegasrec }  
  seek(gf, recno - 1);  
  gf := grec;  
  put(gf)  
end;
```

此時檔案緩衝變數 **gf**，必須在呼叫標準 PASCAL 常式 **put** 之前，就把記錄指派給它。

APPLE 和 UCSD PASCAL 都有一個 **seek** 常式，上述兩程序就利用 **seek** 常式，來放置檔案指示器，所以 **get** 或 **put** 才能讀或寫一特定記錄（記錄由 0 開始註記之，兩程序中以 **recno-1** 表示之）。本版 PASCAL 可用隨機方式接達磁碟檔，也有一些方法重複 **seek** 的動作。

資料登錄 (Data Entry)

dispgasform 將資料登錄在螢幕顯示：

```
procedure dispgasform;  
( Display gas log entry form )  
  
begin { dispgasform }  
  posstr('Record Number:', margin - 2, 5);  
  posstr('Date:', margin, 7);  
  posstr('Brand:', margin, 9);  
  posstr('Odometer Reading:', margin, 11);  
  posstr('Price (cents/gallon):', margin, 13);  
  posstr('Amount (gallons):', margin, 15);  
  posstr('Total Cost:', margin, 17)  
end;
```

gaslog 主常式內定義變數 **margin**，是爲了使 40 和 80 行螢幕的中央的形式合理，**margin** 可以是一個真正的常數，因爲它是由計算而求得的數，所以它必須是個變數。

• 3 交作輸入 •

我們每隔一行才寫一個資料登錄 (data entry) 項目，這樣的安排，對於 CRT 只有 24 行來說，過於奢侈，所以 **dispgasform** 和 **getgasrec** 需作些微修飾。

油料檔案記錄，取自 **getgasrec**：

```
procedure getgasrec(var newrec, lastrec: gasrec; nrecs: integer);
{ Get new gas record from user }

const
    THRESH = 0.02;          { Maximum relative difference between estimated
                             and entered cost }

var
    estcost: fixed;
    change: boolean;

{-----}
{ Modules to be included here: }
{      datecomp                }
{      ftor                    }
{-----}

begin { getgasrec }
    gotoxy(margin + 13, 5);
    write(nrecs + 1);
    newrec := lastrec;
    change := FALSE;
    repeat
        with newrec do begin
            getdate(fdate, margin + 5, 7, fdate, TRUE);
            if (datecomp(fdate, lastrec.fdate) < 0) and (nrecs > 0) then
                remark('WARNING - before last record's date');
            {$v-}
            getstring(brand, BRANDSIZE, margin + 6, 9, brand, [' '..'^'], FALSE);
            {$v+}
            getfixed(mileage, margin + 17, 11, 0, 9999999, mileage, 1, TRUE);
            if mileage <= lastrec.mileage then
                remark('WARNING - mileage is <= prior mileage');
            getfixed(fuelcost, margin + 21, 13, 1, 3000, fuelcost, 1, TRUE);
            getfixed(gallons, margin + 17, 15, 1, 9999, gallons, 2, change);
            estcost := fuelcost * gallons div 1000;
            getfixed(totcost, margin + 11, 17, 1, 30000, estcost, 2, TRUE);
            if abs(ftor(totcost) - ftor(estcost))/ftor(totcost) > THRESH then
                remark('WARNING - cost disagrees with estimate')
            end;
            change := ask('Any changes?(Y/N):', MAXCRTROW - 1, FALSE, TRUE);
            eraseline(MAXCRTROW - 1)
        until not change
    end;
```

本常式相當直截了當，利用前章描述的輸入工具，唯一的技巧，是 { \$V- } 指引名錄關閉字串長度型態的查核。這是 APPLE PASCAL 的特性之一，汽油品牌在宣告時限制 10

個字元，在理論上 **getstring** 却可回轉一個 80 一字元的結果，然而編譯器承認這些會發生問題，所以視同錯誤，但是選擇 { \$V- }，這種錯誤查核工作忽略不作，由於我們在 **getstring** 查核其長度，限制為 10 個字元，所以非常安全。

getgasrec 使用二個常式，來幫助檢查錯誤。第一是 **datecomp** 比較兩個日期，它根據第一與第二兩日期比較結果來回轉一個值，小於是 -1，相等為 0，大於是 +1，該函數如下：

```
function datecomp(d1, d2: date): integer;
{ Compare two dates, return -1,0,1 if d1 is <,> d2 }
begin { datecomp }
  if d1.year < d2.year then
    datecomp := -1
  else if d1.year > d2.year then
    datecomp := 1
  else if d1.month < d2.month then
    datecomp := -1
  else if d1.month > d2.month then
    datecomp := 1
  else if d1.day < d2.day then
    datecomp := -1
  else if d1.day > d2.day then
    datecomp := 1
  else
    datecomp := 0
end;
```

第二個常式是 **ftor** 函數，它將一個定數轉換成一實數：

```
function ftor(f: fixed): real;
{ Convert fixed to real }
var
  new: fixed;
begin { ftor }
  if f < 0 then
    ftor := -ftor(-f)
  else if f <= MAXINT then
    ftor := trunc(f)
  else begin
    new := f div MAXINT;
    ftor := trunc(f - new * MAXINT) + MAXINT * ftor(new)
  end
end;
```

• 3 交作輸入 •

當 **getgasrec** 偵測到可能的錯誤，該錯誤由 **remark** 程序產生報告：

```
procedure remark(rem: string);
{ Put a message on the screen }

begin { remark }
  crt(BEEP);
  center(rem, MAXCRTROW - 1);
  wait;
  eraseline(MAXCRTROW - 1)
end;
```

利用 **remark** 註記一個警告訊息或指令，使用 **wait** 程序顯示訊息：**press <RETURN> to continue** 且等使用者按 **<RETURN>** 鍵：

```
procedure wait;
{ Wait until the user hits the return key }

var
  ch: char;

begin { wait }
  crt(BEEP);
  center('Press <RETURN> to continue', MAXCRTROW);
  ch := getkey(ch, [chr(13)], FALSE);
  eraseline(MAXCRTROW)
end;
```

產生報表 (Report Generation)

唯一留待設計的是，由 **gasreport** 程序排印報告：

```
procedure gasreport(var gf: gasfile; nrecs: integer);
{ Gas usage report }

const
  HDRSIZE = 132;      { Maximum length of report header }
  LPP = 66;           { Lines per printed page }

type
  header = string[HDRSIZE];
```

```

var
  pageno, lineno, rmarg, lmarg, i: integer;
  p: text;
  title, s: string;
  today: date;
  hdr1, hdr2: header;
  grec: gasrec;
  mpg: fixed;

(-----)
( Modules to be included here: )
(   initprinter                 )
(   checkhead                   )
(   checkfoot                   )
(-----)

begin { gasreport }
  crt(CLEAR);
  title := 'Gasoline Usage Report';
  center(title, 0);
  posstr('Today's Date?', margin, 12);
  getdate(today, margin + 14, 12, today, FALSE);
  pageno := 1;
  lineno := 1;
  initprinter(p);
  hdr1 :=
    '          Odometer   Fuel Cost   Amount   Total';
  hdr2 :=
    ' Date      Brand      Reading (cents/gal) (gals)   Cost   MPG';
  lmarg := 5;
  rmarg := 75;
  for i := 1 to nrecs do begin
    checkhead(p, title, '', '', hdr1, hdr2, lmarg, rmarg, today, pageno,
      lineno, 1);

    readgasrec(gf, i, grec);
    with grec do begin
      dtos(fdate, s);
      write(p, ' ': lmarg, s, ' ': 2, brand, ' ': 12 - length(brand));
      ftos(mileage, 8, 1, s);
      write(p, s);
      ftos(fuelcost, 10, 1, s);
      write(p, s);
      ftos(gallons, 11, 2, s);
      write(p, s);
      ftos(totcost, 8, 2, s);
      write(p, s);
      if (i = 1) or (gallons <= 0) or (mileage <= lastrec.mileage) then
        writeln(p, ' ': 4, '----')
      else begin
        mpg := (((1000 * (mileage - lastrec.mileage)) div gallons) + 5) div 10;
        ftos(mpg, 8, 1, s);
        writeln(p, s)
      end
    end;
    lineno := lineno + 1;
    lastrec := grec;
    checkfoot(p, '', lmarg, rmarg, pageno, lineno, LPP - 4)
  end;
  checkfoot(p, '', lmarg, rmarg, pageno, lineno, 2);
  close(p)
end;

```

• 3 交互輸入 •

流程里程表減去前次里程表，再除以購買的加侖數，其商就是每加侖跑的里程，當然，只有每次加油時，油箱內汽油都是一樣多，這種計算才精確。

initprinter 定列表機的初值，以便自程式那接收資料，使用人爲了確定列表機待印報告，要求在定初值之前先「查核」列表機。

```
procedure initprinter(var p: text);  
( Initialize printer )  
  
begin ( initprinter )  
    remark('Please check printer');  
    rewrite(p, 'PRINTER:');  
end;
```

由於標準 PASCAL 並不將輸出的資料視同本文檔案，所以 **initprinter** 就和許多輸入／出常式一樣，不具易傳性。本版程序適用於 APPLE 和 UCSD PASCAL；如用其他版本時，必須寫一個 **initprinter** 使得本文檔 P 和電子計算機的列表機接觸。

一個較複雜版本的 **initprinter** 是，讓使用人選擇報表顯示在 CRT 上，或是把它寫到磁碟檔並付印之，或是把報表列印之。

在 **gaslog** 的分頁報表形式，適用於其他程式。針對列表機的寬度，來定左、右的邊緣。三行抬頭 (title) 被放到每頁的中央，抬頭可能少一行，或是全部沒有，報表日期和頁次，印在每頁第一行的右方，保留二列，作爲「行標題」之用，每頁最下方預留一行。

每行在列印之前，**gasreport** 呼叫 **checkhead**，如果流程行數小於或等於程式加註的值，即列抬頭和標題行：

• 高等 Pascal 程式設計技巧 •

```
procedure checkhead(var p:text; ttl1, ttl2, ttl3:string; hdr1, hdr2:header;
                    lmarg, rmarg: integer; today: date;
                    var pageno, lineno: integer; checklin: integer);
{ Check for header }

var
  s: string;
  i, nb: integer;

begin { checkhead }
  if lineno <= checklin then begin
    writeln(p);
    dtos(today, s);
    nb := lmarg + (rmarg - lmarg - length(ttl1)) div 2;
    writeln(p, ' ': nb, ttl1, ' ': rmarg - nb - length(ttl1) - 8, s);
    nb := lmarg + (rmarg - lmarg - length(ttl2)) div 2;
    writeln(p, ' ': nb, ttl2, ' ': rmarg - nb - length(ttl2) - 8, 'Page ',
                                                    pageno);

    nb := lmarg + (rmarg - lmarg - length(ttl3)) div 2;
    writeln(p, ' ': nb, ttl3);
    writeln(p);
    writeln(p, ' ': lmarg, hdr1);
    writeln(p, ' ': lmarg, hdr2);
    write(p, ' ': lmarg);

    for i := lmarg to rmarg do
      write(p, '-');
    writeln(p);
    lineno := lineno + 8
  end
end;
```

checkfoot 程序設定頁底的格式：

```
procedure checkfoot(var p: text; footer: string; lmarg, rmarg: integer;
                    var pageno, lineno: integer; checklin: integer);
{ Check for footer }

begin { checkfoot }
  if lineno >= checklin then begin
    while lineno < LPP - 3 do begin
      writeln(p);
      lineno := lineno + 1
    end;
    writeln(p, ' ': lmarg + (rmarg - lmarg - length(footer)) div 2, footer);
    lineno := lineno + 1;
    while lineno <= LPP do begin
      writeln(p);
      lineno := lineno + 1
    end;
    lineno := 1;
    pageno := pageno + 1
  end
end;
```


這些常式，全部需要 **gaslog**，我們看了許多 PASCAL 代碼，這些結果都是有用的，我們也寫了一工作程式，也建立了一些工具，這些對於以後的程式，都很有用。

建議 (Suggestions)

改進 **gaslog** 的方法很多，有些很容易辦到，有的却很困難。

最明顯的步驟，是增加報表資訊。許多統計數目，由資料檔推演而來：每哩成本，油箱容量，一箱油跑多少哩，每天哩程等等。你可能對於計算不同品牌的 MPG 平均數很有興趣；如何才累積大量的不同的品牌，是需技巧的部份。

這些資訊全部計算好了，使用人可任意選擇其所需的資料去排印之。

另一種可能是，使用人可用更彈性的方法，鍵入資料，如果前次鍵入資料有錯誤，允許刪除、修訂之。鍵入資料的同時可加上分類資料檔的選擇，而除去按時間次序鍵入資料的限制。

推薦閱讀 (Recommended Reading)

gaslog 的概念來自 Etudes for programmers (C. Watherell 著) 的一個練習題。

4

打碎數字：一般目的計算器

(*Crunching Numbers: A General-Purpose Calculator*)

本章將建立一個「自使用人那接受算術敘述，並且計算它們的值」之程式 **calc**，首先描述該程式的工作。

當 **calc** 開始時，螢幕除了它的上方盒內的程式名稱，和接近底端一個底線字元的字串之外，清除之。和以前一樣，底線字元指示 **calc** 內的資料登錄欄。

calc 在輸入字串方面的工作，我們將稱為敘述 (statements)，最簡單的一種敘述，是一個算術式，例如：

$2 + 3$

當使用人按 <RETURN> 鍵時終止輸入，在敘述之後，**calc** 立即顯示結果。(注意：**calc** 的回答均劃有底線) 例如：

$2 + 3 = \underline{5.00}$

上述結果顯示之後，**calc** 向上捲一行，並且詢問其他敘述，捲動過程中，允許把最近計算的結果留在螢幕之上。

calc 可執行加、減、乘、和除。例如：

• 高等 Pascal 程式設計技巧 •

$$\begin{array}{rcl} 2 + 7 & = & 9.00 \\ 2 - 7 & = & -5.00 \\ 2 * 7 & = & 14.00 \\ 2 / 7 & = & 0.29 \end{array}$$

這些基本的運算，可以混合組成一個單一敘述，如下：

$$5.3 + 13.4 * 1.2 - 19.6/5.8 = \underline{18.00}$$

這些運算有先後的次序，與大多數的代數程式語言相同：乘和除優先，後是加、減。運算次序可因加上括號而改變先後次序；在括號內的運算最優先。例如：

$$\begin{array}{rcl} 1 + 2 * 3 + 4 & = & 11.00 \\ (1 + 2) * (3 + 4) & = & \underline{21.00} \end{array}$$

敘述可以很複雜，例如：

$$3 + 1/(7 + 1/(15 + 1/(1 + 1/293))) = \underline{3.14}$$

除了執行簡單的計算之外，`calc` 尚可選擇把結果儲存到一度數內，供進一步的計算，例如：

$$\begin{array}{rcl} \text{APPLE} & = & 5/6 = \underline{0.83} \\ \text{ORANGE} & = & 1/2 = \underline{0.50} \\ \text{APPLE} + \text{ORANGE} & = & \underline{1.33} \\ (\text{APPLE} - \text{ORANGE})/(\text{APPLE} + \text{ORANGE}) & = & \underline{0.25} \end{array}$$

此處定義兩度數，APPLE 和 ORANGE。度數名字或識別字可以是 1 到 8 個字元（事實上有時可超過 8 字元，但 `calc` 對於額外的字元，視同無意義）。識別字（`identifier`）的第一個字元，必須是字母；其餘的字元，可以是數位或是字母。

使用人可以在一敘述內定義一個（以上）度數，例如

$$\text{PI} = (\text{NUMER} = 355)/(\text{DENOM} = 113) = \underline{3.14}$$

• 4 打碎數字：一般目的計算器 •

本敘述定義三度數：PI，NUMER，和DENOM。

使用人可因敘述的隱藏格式指引（formatting directive），而改變數目顯示的格式，如下：

```
[F6] 10/7 = 1.428571  
[S3] 1.928 * 6.792 = 1.309e1
```

格式指引包含在中括號內，使用人可以將其任意放置。該指引包含一字母和一數字，字母＜F＞是定點格式，而＜S＞是科學格式，數字部份在小數點後設定位數，例如：

```
[F0] A = 5/3 = 2  
[F1] A = 1.7  
[F2] A = 1.67  
[F3] A = 1.667  
...  
[F13] A = 1.6666666666667
```

小數點後面的位數合法範圍是 0 到 13；calc 中數字的精度是 14 數位，數字如果太小或太大，無法用定點格式，calc 即以科學格式顯示結果，在小數點之後，可用 13 數位表示之。

錯誤復原（Error Recovery）

鍵入的敘述句中，只能偵測出二種錯誤，第一，在正確的地方無法看到右括號，以與左括號平衡。（所謂正確的地方意義不清；我們將討論 calc 期望右邊有一個括號。）第二，參考一個未定義的識別號。這些都不是重要的錯誤；如果少一括號，calc 就希望有一個括號。如果參考一未定義的識別號，calc 假定該度數值是 0。如果二種錯誤都有，calc 即向使用人報告錯誤，並繼續估計敘述。

有三種關鍵性錯誤：溢位（overflow）、超下限（under

flow)、和除以 0，在計算時才會有錯，它並不導致程式破壞數字，只是終止計算，並向使用人報告錯誤。

除這些情況之外，縱然敘述包括符號 (symbols) 和數字的隨機組合，**calc** 嘗試使輸入敘述有意義，**calc** 掃描輸入敘述，直到不能進行為止，然後把它的數字化結果顯示，這是很平常的設計，而非文法上的不必要限制，我們盡力去猜使用人的意圖。只要設計的有意義，總是對的。在最差的情況下，**calc** 只回轉一個無意義的答案；它並不崩潰。

資料結構 (Data Structures)

calc 內的數值，並非由 PASCAL 的內建實數資料型態代表。我們使用一個程式定義的「廣域實數」(" extended real ") 資料型態 **xreal**。

爲了達到 **calc** 的目的，產生了兩個不利的結果。第一、實數無論是範圍和精確度都有限制。這個限制未必是任何的應用都適用，在 **calc** 內，程式本身就定義這些限制，我們是在控制，而非語言實行，如需要更大 (小) 的範圍、或更大 (小) 的精度，只改程式即可。

第二、使用內建實數時，很難處理溢位和超下限的錯誤。有些版本的 PASCAL 提供一個非標準機構 (non-standard mechanism) 去捕捉這些錯誤。而 APPLE PASCAL 則不，它很難預期何時會發生超下限、或溢位錯誤，這所謂預期包括在記憶體中、或在計算方法中的實數方面的知識。

在設計 **xreal** 型態的第一步驟，是定義我希望它代表的精確度或數字範圍。選擇 **xreal** 精確數位爲 14，**xreal** 的最小正值是 $1.0e-65$ ，最大正值是 $9.9999999999999e62$ 。通常我們希望定義一個「有用的」範圍和精確度，可適用於大部份的

• 4 打碎數字：一般目的計算器 •

現實世界。我們也喜歡設計 **xreals**，使其在記憶內有效的儲存、接達得非常快。

xreal 由二個數組成：(1)分數 (fraction) 部份，它是一個長整數，可確實掌握我們所希望的精度。(2)指數 (exponent) 部份，是一個平常整數，它可表示 10 的乘方個數。這二個數組成一個真正的數。這是在記憶內表示定點實數的特殊方法。

這個結構用 PASCAL 表示如下：

```
<type>
  xreal = record
    frac: fixed;
    expo: MINEXP..MAXEXP
  end;
```

此處 **MINEXP** 和 **MAXEXP** 分別是指數部份 (exponent part) 的最小和最大值。**fixed** 型態定義同第三章：

```
<type>
  fixed = integer[FSIZE];
```

此處的 **FSIZE** 被定義成 14：

```
<const>
  FSIZE = 14;
```

注意 **FSIZE** 的值與第三章時設定的值不同。

把一數轉換成一 **xreal**，其包含分數 (frac) 和指數 (expo) 兩部份，例如 π 值：

$\pi = 3.14159265358979323846264338327950288...$

第一步驟是以科學符號表示數字，形式如下：

$\langle \text{fpart} \rangle e \langle \text{epart} \rangle$

$\langle \text{fpart} \rangle$ 和 $\langle \text{epart} \rangle$ 都經調整使得

$$0.1 \leq \langle \text{fpart} \rangle < 1.0$$

假定 $\langle \text{fpart} \rangle$ 被捨入成 **FIXSIZE** 數位 (digits) ,
則 π 就變成

$$\text{pi} = .31415926535898e1$$

而分數部份由 $\langle \text{fpart} \rangle$ 去掉小數點而成，於是產生一個
14 一數位整數：

$$\text{pi.frac} = 31415926535898$$

最後，**xreal** 的分數部份是 $\langle \text{epart} \rangle$ 加上一個常數
EXCESS.EXCESS 選成 64：

$$\text{pi.expo} = \langle \text{epart} \rangle + \text{EXCESS} = 65$$

雖然隨意取 **EXCESS** 的值，使得 **xreal** 變數的指數部份
不為負數。**EXCESS** 的值是根據最小或最大正 **xreal** 而定。
為了明白它的工作，如前狀況一樣，**xreal** 的最小正值是

$$\text{minx} = 1.0e-65$$

這個值轉換成一個 **xreal**。第一個步驟，先將 **minx** 的小
數點移位：

$$\text{minx} = .100000000000000e-64$$

使得 **frac** 部份變成

$$\text{minx.frac} = 100000000000000$$

而 **expo** 是由 10 的乘方加 **EXCESS** 而計算出的：

$$\text{minx.expo} = -64 + \text{EXCESS} = 0$$

這蘊含著，我們選定 **EXCESS** 後，最小的指數 (**MINEXP**) 是 0，我們可求得 **MAXEXP** 之值 (最大指數)，計算與求 **xreal** 的最大值相似：

$$\text{maxx} = 9.999999999999999\text{e}62 = .9999999999999999\text{e}63$$

$$\text{maxx.frac} = 999999999999999$$

$$\text{maxx.expo} = 63 + \text{EXCESS} = 127$$

當然可將 **xreal** 資料型態，代表相同範圍的負值。**frac** 的負值指示一個負數。

合法的 **xreal** 變數，總是被正常化的；**frac** 值有 **FIXSIZE** 精確度的數位。唯一例外是 0，它的分數值為 0，而指數值是 **MINEXP**。

變數儲存 (Variable Storage)

任何人使用 **calc** 時，都可定義任意數的變數，其名字可任意定之。我們希望 **calc** 能檢索 (retrieve) 前面已定義的變數值，指派一數給該變數，並且在該變數名字之下儲存，以取代該變數的舊值。更有甚者，我們希望這個運算的速度快，即使有許多變數使用中，亦不例外。

許多儲存和檢索的方法，都可用來代表記憶體內的變數。茲選定二分樹 (binary tree) 為例。許多教科書均描寫二分樹，如讀者不熟悉它的用法，可參考本章最後的書目 (bibliography)。

二分樹的每一節 (node) 都包括變數名字 **xreal** 值，和

• 高等 Pascal 程式設計技巧 •

左指標、右指標指著該結下的子樹 (subtrees) 。本資料很容易在 PASCAL 裏建立；可定義如下：

```
<type>
node = record
  name: identifier;
  value: xreal;
  left, right: nodeptr
end;
```

nodeptr在前面已定義成一個節 (node) 的指標：

```
<type>
nodeptr = ^node;
```

而 **identifier** 型態已定義成一字串，最大長度為 8：

```
<const>
IDSIZE = 8;

<type>
identifier = string[IDSIZE];
```

除了這些主要的資料結構之外，尚有三個次要的資料型態，**register**，**xresuer**，和 **format**。

• 我們定義 **register** 資料型態，為一可以掌握 30 個數位的長整數。

```
<const>
REGSIZE = 30;

<type>
register = integer[REGSIZE];
```

在 **calc** 中，利用 **register** 型態變數，掌握計算的結果。如果增減 **xreal** 的精確度，就必須也增、減 **register** 資料型態的長度。在本章後段要發展的算術常式需要 **register** 變數，掌握 $2 \times \text{FIXSIZE} + 2$ 數位。

因為 PASCAL 的長整數數位最大為 36，則

- 4 打碎數字：一般目的計算器 •

FIXSIZE的上限為 17（因 $2 \times 17 + 2 = 36$ ）。由於應用方面的需求，可以用一較小的 **register** 來修飾演算法（**algorithm**）或自行設計「長整數」資料型態，而讓本型態受到較少的限制（見附錄 A）。

- 利用 **xresult** 資料型態的變數，來代表計算的狀況：

```
<type>
xresult = (OK, OVERFLOW, UNDERFLOW, ZERODIVIDE);
```

通常的狀況是 **OK**，然而程式偵測到溢位、超下限、或除以 0，該狀態設定成適當值。

- 最後以 **format** 變數型態，掌握 **calc** 的流程格式

```
<type>
format = (FIXEDPOINT, SCIENTIFIC);
```

如果格式是 **FIXEDPOINT**，程式就以定點格式顯示結果，如格式是 **SCIENTIFIC**，則顯示格式，就是科學記號。小數點後面的有效位數，就由全盤 (**global**) 整數變數 **ndigs** 掌握之。

計算的主常式 (The Main Routine)

根據前面的描述，主常式虛擬代碼如下：

```
begin
  initialize
  repeat
    while the user's input statement is non-null
      extract formatting directive(s), if any
      evaluate statement
      if no fatal errors
        display result
      else
        display error message
  scroll CRT
```

• 高等 Pascal 程式設計技巧 •

```

        end-while
        ask if user is done
    until user is done
end

```

翻譯成 PASCAL :

```

($s+)
program calc;
{ Calculator Emulator }

const
    MAXCRTCOL = 79;      { maximum CRT column }
    MAXCRTROW = 23;      { maximum CRT row }
    MAXSTR = 80;         { maximum string length }
    EXCESS = 64;         { xreal exponent excess }

    MINEXP = 0;          { minimum xreal exponent }
    MAXEXP = 127;        { maximum xreal exponent }
    FIXSIZE = 14;        { maximum digits in fixed number }
    REGSIZE = 30;        { max digits in register number, 2 * FIXSIZE + 2 }
    IDSIZE = 8;          { maximum identifier length }

type
    crtcommand = (HOME, CLEAR, ERASEOL, ERASEOS, UP, DOWN, LEFT, RIGHT, BEEP);
    charset = set of char;
    fixed = integer[FIXSIZE];
    xreal = packed record
        expo: MINEXP..MAXEXP;
        frac: fixed
    end;
    register = integer[REGSIZE];
    xresult = (OK, OVERFLOW, UNDERFLOW, ZERODIVIDE);
    format = (FIXEDPOINT, SCIENTIFIC);
    identifier = string[IDSIZE];
    nodeptr = ^node;
    node = record
        name: identifier;
        value: xreal;
        left, right: nodeptr
    end;

var
    alldone: boolean;
    s, xstr: string;
    valid: charset;
    i, ndigs, inpline: integer;
    x: xreal;
    result: xresult;
    root: nodeptr;
    ten: array [0..REGSIZE] of register;
    fmt: format;

{-----}
{ Modules to be inserted here: }
{ crt }
{ posstr }
{ getkey }
{ addchar }
{ chopchar }
{ getstring }
{ eraseline }

```

• 4 打碎數字：一般目的計算器 •

```
(      center              )
(      getboolean          )
(      ask                 )
(      gchar               )
(      gnbchar             )
(      stof                )
(      ftos                )
(      wait                )
(      remark              )
(      initcalc            )
(      xnorm               )
(      xadd                )
(      xtos                )
(      evalstmt            )
(      crtscroll           )
(      disptitle           )
(      getstmt             )
(      findformat          )
(-----)
```

```
begin { calc }
  crt(CLEAR);
  disptitle('calc');
  initcalc;
  repeat
    while getstmt(s) do begin
      findformat(s, fmt, ndigs);
      posstr('= ', length(s) + 1, inpline);
      i := 1;
      result := evalstmt(s, i, x);
      if result = OK then
        result := xtos(x, fmt, ndigs, xstr);
      if result = OK then
        posstr(xstr, length(s) + 3, inpline)
      else begin
        posstr('?', length(s) + 3, inpline);
        case result of
          OVERFLOW:
            remark('Sorry: overflow');
          UNDERFLOW:
            remark('Sorry: underflow');
          ZERODIVIDE:
            remark('Sorry: division by zero')
        end
      end;
      crtscroll(1);
      disptitle('calc')
    end;
    alldone := ask('Exit?(Y/N): ', MAXCRTROW - 1, FALSE, FALSE);
    eraseline(MAXCRTROW - 1)
  until alldone;
  crt(CLEAR)
end.
```

calc 使用在前二章發展的 16 個程式和函數。我們開始認識模組化 (modular) 的優點：它減少設計程式的工夫，並

使程式之外觀和作用更一致。

全盤變數的初值化 (Initializing Global Variables)

大部份的程式包括少許可以接達整個程式的全盤變數 (global variables)。習慣上我以個別的常式，來定這些變數的初值，該常式名叫 **initcalc**：

```
procedure initcalc;  
( Initialize global variables )  
  
var  
    i: integer;  
  
begin ( initcalc )  
    valid := ['A'..'Z', '0'..'9', '+', '-', '*',  
             '/', ' ', '.', '=', '(', ')', '[', ']'];  
    root := NIL;  
    ten[0] := 1;  
    for i := 1 to REGSIZE do  
        ten[i] := 10 * ten[i - 1];  
    fmt := FIXEDPOINT;  
    ndigs := 2;  
    inpline := MAXCROW - 3;  
end;
```

initcalc 定下列變數的初值：

- | | |
|-------------------|----------------------------------------------------------------------------|
| Valid | 使用人在一輸入敘述中，鍵入字元集。 |
| root | 二分樹 (binary tree) 的根節點指標，其儲存在變數裏，初值定為 NIL ， NIL 指示該樹最初是空的。 |
| ten | 一個 10 的乘幂陣列； $ten[1] = 10$ ， $ten[6] = 1000000$ ，等等，該陣列的元素在程式內用作不同的計算。 |
| fmt, ndigs | 顯示的數字之格式，最初定為在小數點之後有 2 個數位。 |
| inpline | 使用人在 CRT 鍵入敘述之處。 |

三個簡易的模組 (Three Easy Modules)

calc 需要三個簡單的模組 (modules)，第一是 **crtscroll** 程序，它可捲動 CRT 螢幕若干行：

```
procedure crtscroll(n: integer);  
( Scroll CRT n lines )  
  
begin ( crtscroll )  
  gotoxy(0, MAXCRTROW);  
  while n > 0 do begin  
    writeln;  
    n := n - 1  
  end  
end;
```

第二，**disptitle** 常式將程式的名稱，放到螢幕頂端的盒子：

```
procedure disptitle(s: string);  
( Display title in box at top of screen )  
  
var  
  i, nch: integer;  
  
begin ( disptitle )  
  
  eraseline(0);  
  eraseline(2);  
  center(s, 1);  
  crt(RIGHT);  
  write('*');  
  crt(DOWN);  
  nch := length(s) + 4;  
  for i := 1 to nch do begin  
    crt(LEFT);  
    write('*');  
    crt(LEFT)  
  end;  
  crt(UP);  
  write('*');  
  crt(LEFT);  
  crt(UP);  
  for i := 1 to nch do  
    write('*')  
end;
```

最後，**getstmt** 常式自使用人那兒，取得輸入的敘述：

```
function getstmt(var s: string): boolean;  
( Get statement from user )  
  
begin ( getstat )  
  getstring(s, MAXCRTCOL - FIXSIZE - 8, 0, inpline, '', valid, TRUE);  
  getstmt := (length(s) > 0)  
end;
```

我們設定使用人的輸入字串的最大長度為 **MAXCRTCOL - FIXSIZE - 8**，它使得輸入敘述和結果，可在 **CRT** 上相關之列。**calc** 顯示一個等號，並在輸入敘述後立即接著數值的結果，大約需要 23 個字元位置。

這種方法，限制 40 行輸入敘述的有效空間，它的顯示像蘋果二號，如設定 **MAXCRTCOL** 是 39，則使用人用作輸入字元的位置，就只有 17 個字元。若如是，你可能希望：(1)修飾顯示程式，把結果放在下一行，並增加最大的長度，充分使用全螢幕。或(2)設定 **MAXCRTCOL** 為 79，利用蘋果二號的螢幕耦合特性，把螢幕分割成二半。

getstmt 利用 **getstring** 的軟體的鎖住移位特性，把所有輸入的字母，變為大寫字母，如此這般，減輕許多複雜性，把容易混淆的可能性變為最小，把識別符號 **APPLE**、**apple**、**AppLe** 或 **aPpLe** 看作相同之變數。當寫變數的儲存體和檢索常式時，不必擔心這些問題，我們在輸入時就已解決了。

當使用人簡單的按 **<RETURN>** 而不輸入敘述時，**getstmt** 回復 **FALSE** 作為函數結果。否則它回復 **TRUE**。

摘取格式指引 (Extracting Formatting Directive)

像前面的綱要一樣，在評估敘述之前，**calc** 即摘取格式的指引，把格式指引和敘述評估混合解釋，可以使得程式複雜。當 **calc** 在輸入敘述中找或解譯每一格式指引時，用空白來取代之；這個原理，確保程式的一部份作算術運算時，不必知

道格式指引。同理，格式指引解譯器，也無須知道算術。

findformat 設計如下：

```
procedure findformat(var s: string; var fmt: format; var ndigs: integer);
  ( find, extract, and erase formatting directives in string )

var
  i, j: integer;
  c: char;
  f: fixed;

begin ( findformat )
  i := 1;
  while i <= length(s) do begin
    while not (gnbchar(s, i, c) in ['E', chr(0)]) do
      i := i + 1;
    if c = 'E' then begin
      j := i;
      i := i + 1;
      if gnbchar(s, i, c) = 'F' then begin
        fmt := FIXEDPOINT;
        i := i + 1;
      end
    else if c = 'S' then begin
      fmt := SCIENTIFIC;
      i := i + 1;
    end;
    if gnbchar(s, i, c) in ['0'..'9'] then
      if stof(s, i, 0, f) then
        if (f >= 0) and (f <= FIXSIZE - 1) then
          ndigs := trunc(f);
    while not (gnbchar(s, i, c) in ['J', chr(0)]) do
      i := i + 1;
    if c = 'J' then
      i := i + 1;
    while j < i do begin
      s[j] := ' ';
      j := j + 1;
    end
  end
end
end
end;
```

敘述語法 (Statement Syntax)

我們已用描象的項目「敘述」來描述 **calc** 字串之值，我們也舉出許多這種敘述的例子。當我們設計部份程式時，我們必須精確的描述敘述。

語法 (syntax) 圖表，是一種最自然，最易了解合法的

敘述語法。大部份的介紹 PASCAL 語言的教科書，就是用語法圖來描述該語言。對於任何方面的應用，包括語言分析，語法是最有用的工具，**calc** 亦不例外。

圖 4-1 是一敘述的語法圖，其解釋非常簡單。每一圖有一個起始點和尾點，箭頭表合法的方向（單行道）。岔口代表改變或重複結構。長方形表示在其他語法圖定義的語言元素，圓形展示完全相配的符號。圖 4-1 內的敘述，包含在長方形內，該敘述圖形顯示敘述是可以遞迴的。敘述也可以是一個表式（**expression**）或一字串，字串形式是 **identifier = statement**（我們稱為指派）。

語法圖並未告訴我們如何處理語言元素；它只教我們如何認識元素。

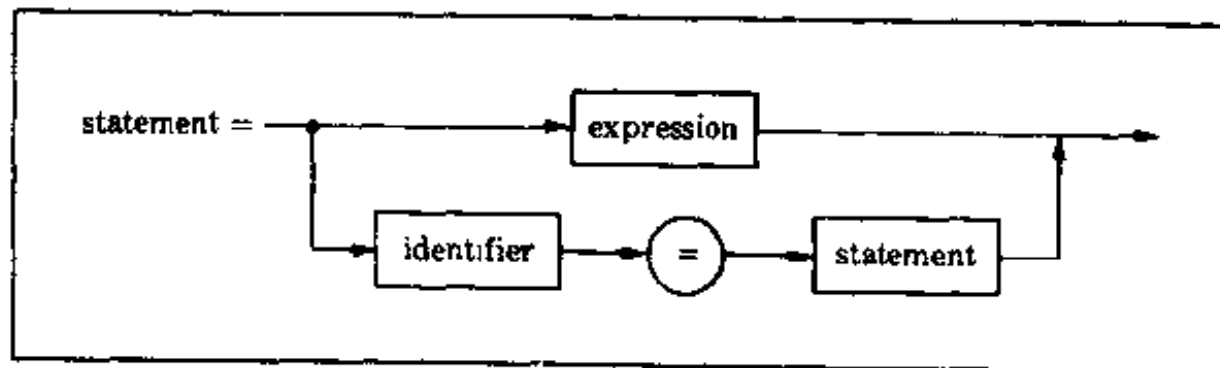


圖 4-1 敘述的語法圖

我們現在知道敘述是怎麼一回事了，可以寫一個常式評估它們，所謂「評估」就是計算數值之意，如果敘述是個指派，我們也能把敘述值，指派給特定的識別符號。

當敘述一開始就是一個識別符號，就比較複雜，雖然我們並未正式定義敘述，我們知道表式像：

APPLE + ORANGE - PEAR

則指派就成為：

• 4 打碎數字：一般目的計算器 •

APPLE = GRAPE - LEMON/LIME + 34

這二者如何分辨呢？如果一敘述的領頭是識別符號，則識別符號後，我們注意那非空白字元，如果是 $\langle = \rangle$ 字元，就有一指派。若該字元不是 $\langle = \rangle$ ，則敘述必然是一表式。（如敘述的領頭不是一識別符號，它必定是一個表式）

利用語法圖，敘述評估常式的虛擬代碼可表示如下：

```
begin
  remember beginning of statement
  if the first thing in the statement is an identifier then
    if the next (nonblank) character is an equals sign then
      skip over equals sign
      evaluate statement from this point
      if calculation OK then
        assign value to identifier
      else
        backtrack - evaluate expression from beginning
    else
      evaluate expression
end
```

evalstmt 函數被呼叫形態如下：

result := evalstmt(s, i, x)

輸入參數 s 是評估 evalstmt 的字串，而變數參數 i 是輸入和輸出參數。在輸入的時候，i 指示 evalstmt 在字串中開始評估的位置；輸出時，它告訴呼叫常式，藉著它指向敘述之後，在字串中下一個非空白字串，的位置而停止評估。var 參數 x 是看到的敘述之數值。最後，將計算狀態當作函數結果回轉。如回轉的狀態是 OK 表示沒有致命錯誤，否則即回轉一個錯誤代碼。評估其他元素時，很技巧的使用這種呼叫序列的技巧，比較複雜的例子，容後再敘。

敘述評估函數以 PASCAL 寫成：

```

function evalstat(var s: string; var i: integer; var x: xreal): xresult;
{ Evaluate input statement }

var
  id: identifier;
  c: char;
  isave: integer;
  result: xresult;

{-----}
{ Modules to be included here: }
{      findid      }
{      evalexpr    }
{      assign      }
{-----}

begin { evalstat }
  isave := i;
  if findid(s, i, id) then
    if gnbchar(s, i, c) = '=' then begin
      i := i + 1;
      result := evalstat(s, i, x);
      if result = OK then
        assign(id, x)
      end
    else begin
      i := isave;
      result := evalexpr(s, i, x)
    end
  else
    result := evalexpr(s, i, x);
  evalstat := result
end;

```

如果 **evalstat** 看到一個表示，就呼叫 **evalexpr** 常式，計算該表式之值。如見到一個指派，記下該識別符號，並呼叫本身，遞迴的計算在等號之後的子敘述（**substatement**）。

認識識別符號 (Identifier Recognition)

下步驟即建一常式，以便識別和回轉識別符號。圖 4-2 即一識別符號的語法圖。

我們並未將元素 **letter** 和 **digit** 語法圖列舉之，我們認為讀者對它們已有深刻的認識了。圖 4-2 與我們在前面描述合法的識別符號相吻合。

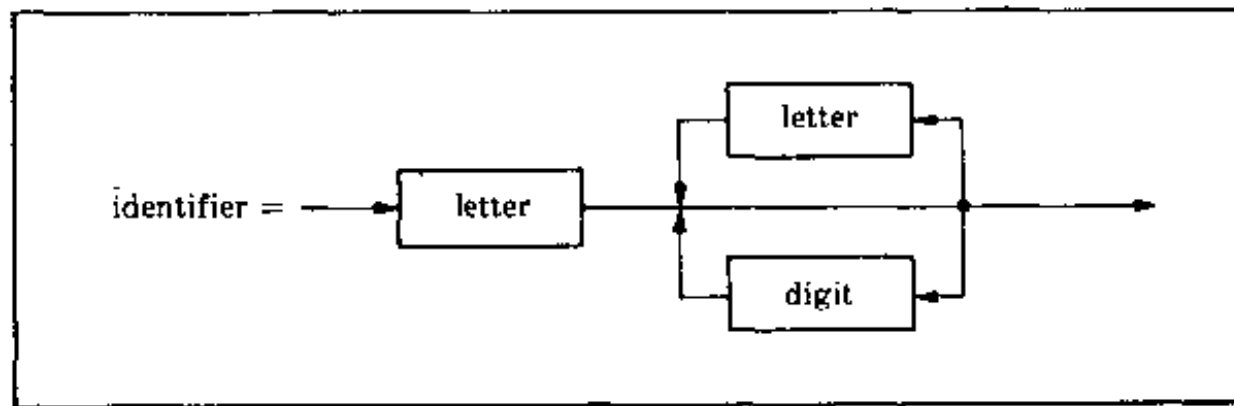


圖 4-2 識別語法圖

findtd 常式注視一識別符號在字串中一特定位置。它由下列方式被呼叫：

```
idseen := findid(s, i, id)
```

如果在字串中特定位置中看到一識別符號，回轉**TRUE**值；離開的時候，變數 **id** 包含識別符號，參數 **i** 即指著，下一個非空白字元。在特定位置找不到識別符號，該函數回轉**FALSE** 值，而 **i** 值不變。

```

function findid(var s: string; var i: integer; var id: identifier): boolean;
(* Get identifier from string *)
var
  c: char;
begin (* findid *)
  id := '';
  if gnbchar(s, i, c) in ['A'..'Z', 'a'..'z'] then begin
    ($v-)
    addchar(id, c, IDSIZE);
    ($v+)
    i := i + 1;
    while gnbchar(s, i, c) in ['A'..'Z', 'a'..'z', '0'..'9'] do begin
      ($v-)
      addchar(id, c, IDSIZE);
      ($v+)
      i := i + 1;
    end;
    findid := TRUE
  end
  else
    findid := FALSE
end;
  
```

findid 常式允許識別符號內附加空白；此即前面提到的，使得 **calc** 更具彈性。例如，使用人員可以鍵入下列敘述

PERIM = SIDE 1 + SIDE 2 + SIDE 3

許多電子計算機語言，不允許識別符號內有空白字元。有很多的理由，使得這些語言，在語法上有嚴格的限制。但是在 **calc** 中，我們盡力寬大。（在本計畫中，是否藏任何缺點嗎？）

表式求值 (Expression Evaluation)

下步驟是求表式之值，表式語法見圖 4-3。圖中表式是一個或多個項目，由正號和負號分開的序列；第一項可能有引導符號。到目前為止，「項目」尚未定義；我們也需要了解語法圖中的項目。

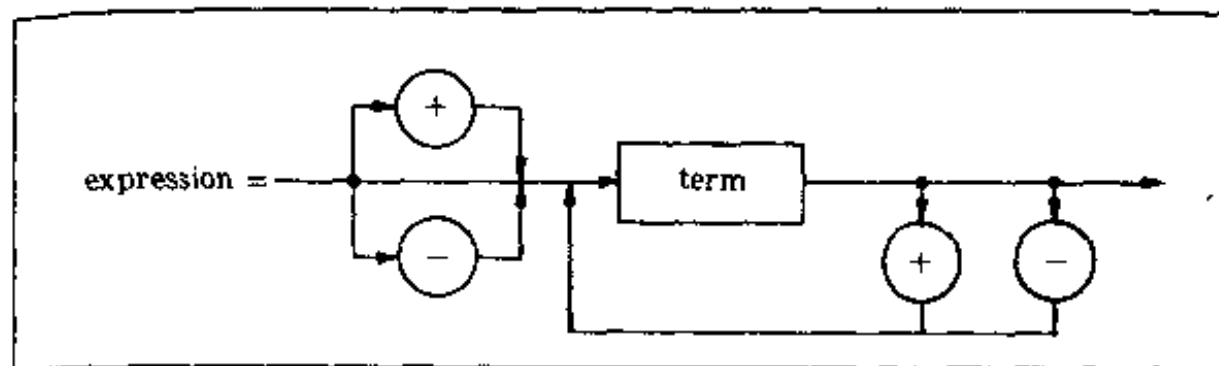


圖 4-3 表示的語法圖

把語法圖，以很自然的態度，翻成 PASCAL 語言：

```

function evalexpr(var s: string; var i: integer; var x: xreal): xresult,
( Evaluate expression )

var
  c: char;
  x1: xreal;
  result: xresult;
  negterm: boolean;

{-----}
{ Modules to be included here: }
{      xsub      }
{      evalterm  }
{-----}
begin ( evalexpr )
  negterm := FALSE;
  if gnbchar(s, i, c) in ['+', '-'] then begin
    i := i + 1;
    negterm := (c = '-')
  end;
  result := evalterm(s, i, x);
  if negterm then
    x.frac := -x.frac;
  while (result = OK) and (gnbchar(s, i, c) in ['+', '-']) do begin
    i := i + 1;
    result := evalterm(s, i, x1);
    if result = OK then
      if c = '+' then
        result := xadd(x, x1, x)
      else ( c = '-' )
        result := xsub(x, x1, x)
    end;
  end;
  evalexpr := result
end;

```

`evalexpr` 首先檢查選擇性引導符號，如引導符號是負號，`negterm` 變數就被指派成 **TRUE** 值。如果沒有引導符號，或者引導符號為正，則 `negterm` 是 **FALSE**。`evalexpr` 於是呼叫 `evalterm` 去求第一項的值。若 `negterm` 是 **TRUE**，第一項的值就無效。第一項就變成和。

其次 `evalexpr` 檢查附加的項目，若下一個非空白字元是一個 `<+>`，或 `<->`，`evalterm` 即評估下一項。在每一項都評估完畢，就加到和去，或自和中減掉。這個算術運算由 `xadd` 和 `xsub` 處理。表式繼續求值，直到再也看不到符號止。

遇到重大錯誤（`evalterm`，`xadd` 或 `xsub` 回轉一個不是 **OK** 的字元）時就終止評估。

項目求值 (Term evaluation)

項目的求值，比表式求值簡單，詳圖 4-4。項目由一個或多個因素 (factor) 組成，因素與因素之間由星號，或斜線分開。把項目翻成 PASCAL 是：

```
function evalterm(var s: string; var i: integer; var x: xreal): xresult;  
{ Evaluate term of expression }  
  
var  
    x1: xreal;  
    result: xresult;  
    c: char;  
  
{-----}  
{ Modules to be inserted here: }  
{     evalfact                     }  
{     xmul                         }  
{     xdiv                         }  
{-----}  
  
begin { evalterm }  
    result := evalfact(s, i, x);  
    while (result = OK) and (gnbchar(s, i, c) in ['*', '/']) do begin  
        i := i + 1;  
        result := evalfact(s, i, x1);  
        if result = OK then  
            if c = '*' then  
                result := xmul(x, x1, x);  
            else { c = '/' }  
                result := xdiv(x, x1, x);  
        end;  
        evalterm := result  
    end;  
end;
```

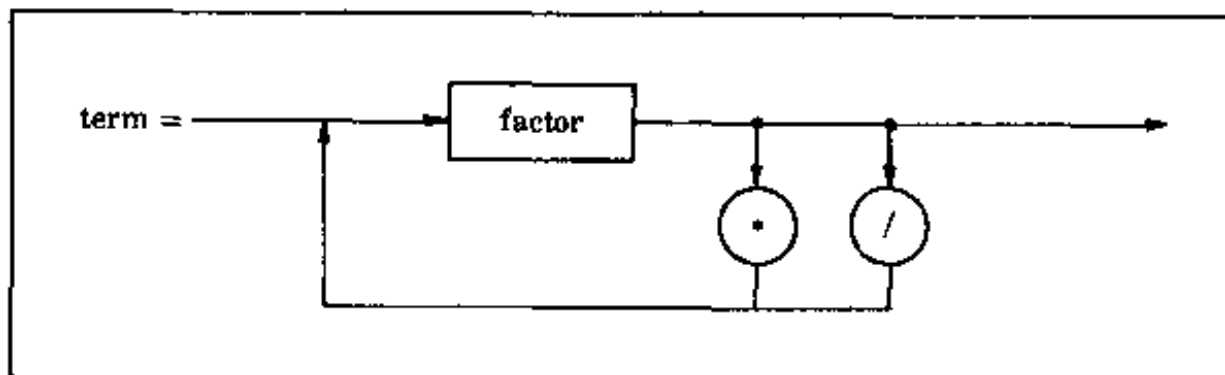


圖 4-4 項目的語法圖

• 4 打碎數字：一般目的計算器 •

再者，`evalterm` 代表計算一特定的語言元素的答案。
`evalterm` 甚至於並不道因素是什麼，做乘和除運算時，交由 `xmul` 和 `xdiv` 處理之。

因素求值 (Factor Evaluation)

其次，求因素之值，語法見圖 4-5。一個因素是包在括號內的一個識別符號，或一個數，也許它有個引導符號，其對應的 PASCAL 如下：

```
function evalfact(var s: string; var i: integer; var x: xreal): xresult;
( Evaluate factor )

var
  c: char;
  negfact: boolean;

{-----}
{ Modules to be inserted here: }
{      stox      }
{      evalid    }
{-----}

begin ( evalfact )
  negfact := FALSE;
  if gnbchar(s, i, c) in ['+', '-'] then begin
    i := i + 1;
    negfact := (c = '-')
  end;
  if gnbchar(s, i, c) = '(' then begin
    i := i + 1;
    evalfact := evalstmt(s, i, x);
    if gnbchar(s, i, c) = ')' then
      i := i + 1
    else
      remark('Missing right parenthesis')
  end
  else if findid(s, i, id) then
    evalfact := evalid(id, x)
  else
    evalfact := stox(s, i, x);
  if negfact then
    x.frac := -x.frac
end;
```

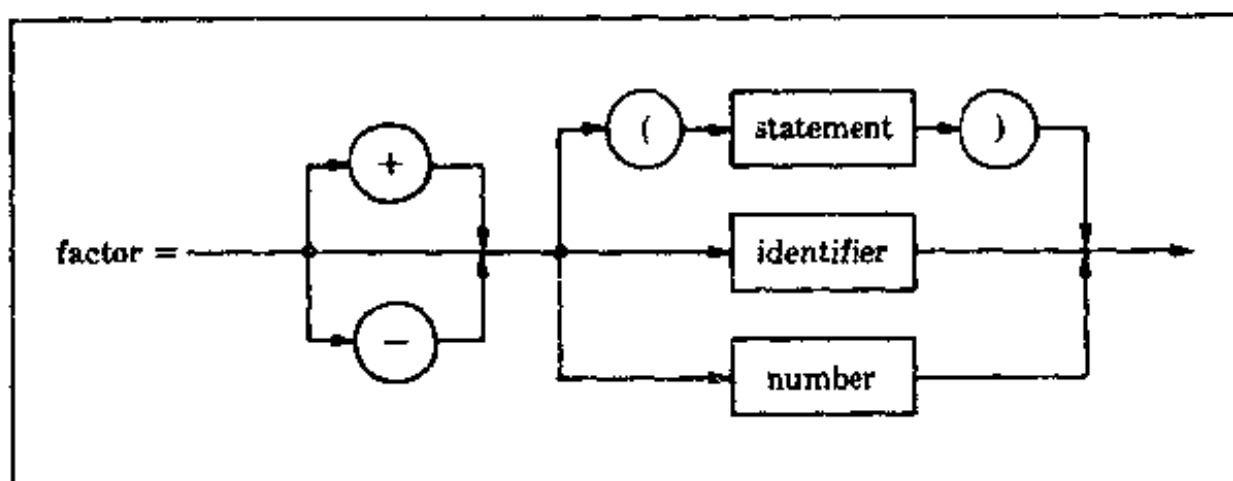


圖 4-5 因素值的語法圖

在注意或略過選擇符號之後，有三個分支路線，**evalfact** 檢視非空白字元。如果它發現一左括號，常式就跳過去，再呼叫 **evalstmt**，以便求後繼敘述的值。如它看到的是一個識別符號，呼叫 **evalid** 求識別符號之值。否則，就呼叫 **stox**，**stox** 嘗試把因素當作一數字。

當 **evalfact** 偵測到「沒有左括號」的錯誤語法，根據語法圖，有一左括號，就必須有一右括號，才能平衡。當於求值過程中，**evalfact** 發現左括號，其就期待右括號出現，如果找到右括號，就略過它；如沒有右括號，就顯示錯誤訊息。

算術 (Arithmetic)

在處理字元字串與實數資料型態之間的轉換之前，茲設常式去實行算術運算，本書不討論、證明該常式，若有興趣探討這方面的問題，請看本章最後的書目。

第一個任務（最困難）是加法。我們需要一個常式，將 2 個廣域實數（**extended real**）相加，並回轉結果。該結果

• 4 打碎數字：一般目的計算器 •

的捨入精度是 **FIXSIZE** 數位，同時也必須知道加法結果是否溢位或超下限。茲列舉 **xadd** 常式如下：

```
function xadd(arg1, arg2: xreal; var x: xreal): xresult;  
( Add two extended reals )  
  
var  
  rawfrac: register;  
  rawexpo: integer;  
  temp: xreal;  
  
begin { xadd }  
  if arg1.expo < arg2.expo then begin { swap }  
    temp := arg1;  
    arg1 := arg2;  
    arg2 := temp;  
  end;  
  if arg1.expo - arg2.expo >= FIXSIZE + 2 then begin { |arg1| >> |arg2| }  
    x := arg1;  
    xadd := OK;  
  end  
  else begin  
    rawexpo := arg1.expo;  
    rawfrac := arg1.frac * ten[FIXSIZE + 1] +  
               arg2.frac * ten[FIXSIZE + 1 - (arg1.expo - arg2.expo)];  
    xadd := xnorm(rawexpo, rawfrac, x)  
  end  
end;
```

開始的時候，務必要使第二個引數（ argument ）的指數小於或等於第一引數的指數，不然就把這兩引數交換，使得滿足前述的規定。

其次，如果兩個引數的指數相差太多，即使把第二個引數加到第一個引數，並不改變第一引數（例如，把電子質量加太陽質量）。這種情形下，我們就把結果設定成第一引數，並結束計算。

通常 **xadd** 處理加法。其目的在於把引數相加後產生一個原始分數，和一原始指數。**xnorm** 常式將這兩個引數組成一個正常的 **xreal** 變數。原始分數（ raw fraction ）是 **register** 型態的一個變數，**register type** 可以掌握的整數數位（ digits ）是 $2 * \text{FIXSIZE}$ ，而原始指數是一整數，但其範圍不受 **MINEXP**..**MAXEXP** 的限制，這兩個組合成一個

「雙倍精密度」(double-precision)，**xnorm** 採捨入並且適合 **xreal** 變數。

爲了加法運算，首先將兩個引數的分數，乘上因素 **ten** [**FIXSIZE+1**] ；其次，將兩引數除以第二個引數的 **frac**，而 **frac** 選擇適當的 10 之乘方，以便整理小數點。最後，兩個 **fraes** 相加，原始指數則設定成第一個引數的指數。

原始分數 **rawfrac** 傳送到 **xnorm**，最多有 $2 \times \text{FIXSIZE} + 2$ 個數位（爲什麼？）。原始分數經捨入後成爲 **FIXSIZE** 數位，而 **xnorm** 據之調整原始指數，此時可測試，由上述兩部份所組合成的廣域實數，看看是否溢位、或超下限，其回轉狀態是 OK 或其他錯誤代碼。

xnorm 的虛擬代碼如下：

```
begin
  if raw fraction is zero
    set xreal to zero and return
  else
    if raw fraction is negative
      change its sign
      remember that it was negative
    end-if
    while raw fraction has fewer than (2 * FIXSIZE + 1) digits
      multiply it by 10 (scale left)
      decrease raw exponent by one
    end-while
    repeat
      if raw fraction has more than (2 * FIXSIZE + 1) digits
        divide by 10 (scale right)
        increase raw exponent by one
      end-if
      round raw fraction to FIXSIZE digits
        (by adding  $5 \times 10^{-\text{FIXSIZE}}$ )
    until raw fraction has (2 * FIXSIZE + 1) digits
    if raw exponent > max exponent
      overflow!
    else if raw exponent < min exponent
      underflow!
    else
      set xreal exponent = raw exponent
      set xreal fraction = raw fraction / ( $10^{-(\text{FIXSIZE} + 1)}$ )
      (xreal fraction will then have FIXSIZE digits, as required)
    end-else
    if original raw fraction was negative
      negate xreal fraction
    end-else
  end
```

• 4 打碎數字：一般目的計算器 •

根據虛擬代碼寫的 PASCAL 如下：

```
function xnorm(rawexpo: integer; rawfrac: register; var x: xreal): xresult;
( Round and normalize extended real )

var
    negfrac: boolean;

begin ( xnorm )
    with x do
        if rawfrac = 0 then begin
            frac := 0;
            expo := MINEXP;
            xnorm := OK
        end
        else begin
            negfrac := (rawfrac < 0);
            if negfrac then
                rawfrac := -rawfrac;
            while rawfrac < ten[2 * FIXSIZE] do begin
                rawfrac := ten[1] * rawfrac;
                rawexpo := rawexpo - 1
            end;
            repeat
                if rawfrac >= ten[2 * FIXSIZE + 1] then begin
                    rawfrac := rawfrac div ten[1];
                    rawexpo := rawexpo + 1
                end;
                rawfrac := rawfrac + 5 * ten[FIXSIZE]
            until rawfrac < ten[2 * FIXSIZE + 1];
            if rawexpo > MAXEXP then begin
                expo := MAXEXP;
                frac := ten[FIXSIZE] - 1;
                xnorm := OVERFLOW
            end
            else if rawexpo < MINEXP then begin
                expo := MINEXP;
                frac := ten[FIXSIZE - 1];
                xnorm := UNDERFLOW
            end
            else begin
                expo := rawexpo;
                frac := rawfrac div ten[FIXSIZE + 1];
                xnorm := OK
            end;
            if negfrac then
                frac := -frac
        end
    end
end;
```

至於在廣域實數中作算術運算的常式，就相當容易。xsub
單單轉化第二個引數的符號，並呼叫 xadd 常式作計算：

• 高等Pascal 程式設計技巧 •

```
function xsub(arg1, arg2: xreal; var x: xreal): xresult;  
{ Subtract two extended reals }  
  
begin { xsub }  
  arg2.frac := -arg2.frac;  
  xsub := xadd(arg1, arg2, x)  
end;
```

乘法常式也簡單，只需呼叫 **xnorm**：

```
function xmul(arg1, arg2: xreal; var x: xreal): xresult;  
{ Multiply two extended reals }  
  
var  
  rawexpo: integer;  
  rawfrac: register;  
  
begin { xmul }  
  rawexpo := arg1.expo + arg2.expo - EXCESS + 1;  
  rawfrac := arg1.frac * arg2.frac;  
  xmul := xnorm(rawexpo, rawfrac, x)  
end;
```

最後，除法常式必須處理除以 0 的狀況：

```
function xdiv(arg1, arg2: xreal; var x: xreal): xresult;  
{ Divide two extended reals }  
  
var  
  rawfrac: register;  
  rawexpo: integer;  
  
begin { xdiv }  
  if arg2.frac = 0 then begin  
    if arg1.frac < 0 then  
      x.frac := 1 - ten[FIXSIZE]  
    else  
      x.frac := ten[FIXSIZE] - 1;  
    x.expo := MAXEXP;  
    xdiv := ZERODIVIDE  
  end  
  else begin  
    rawfrac := ((arg1.frac * ten[FIXSIZE+2]) div arg2.frac) * ten[FIXSIZE-2];  
    rawexpo := arg1.expo - arg2.expo + EXCESS + 1;  
    xdiv := xnorm(rawexpo, rawfrac, x)  
  end  
end;
```

• 4 打碎數字：一般目的計算器 •

除以 0 時，`xdiv` 回轉一個 `ZERODIVIDE` 狀態給呼叫常式，並將 `x` 設定成一個很大的正數或負數 `xread`，正負選擇取決於 `arg1`（在 `calc` 中則無此必要，因為計算會終止，任何除以 0 的結果均被丟棄）。

一個字串轉變成一個廣域實數

(Converting a String to an Extended Real)

現在我建一常式，來轉換一個字串成一廣域實數。其次，我們的目的是要使語法盡可能保持彈性。數的語法圖見圖 4-6。簡單的說，數由下列因子組成：

- 一個選擇性的符號。
- 零或更多的數位。
- 零或更多的數位之後的小數點。
- 包括「E」字母的指數部份，一個選擇性的符號，和零或更多的數位。

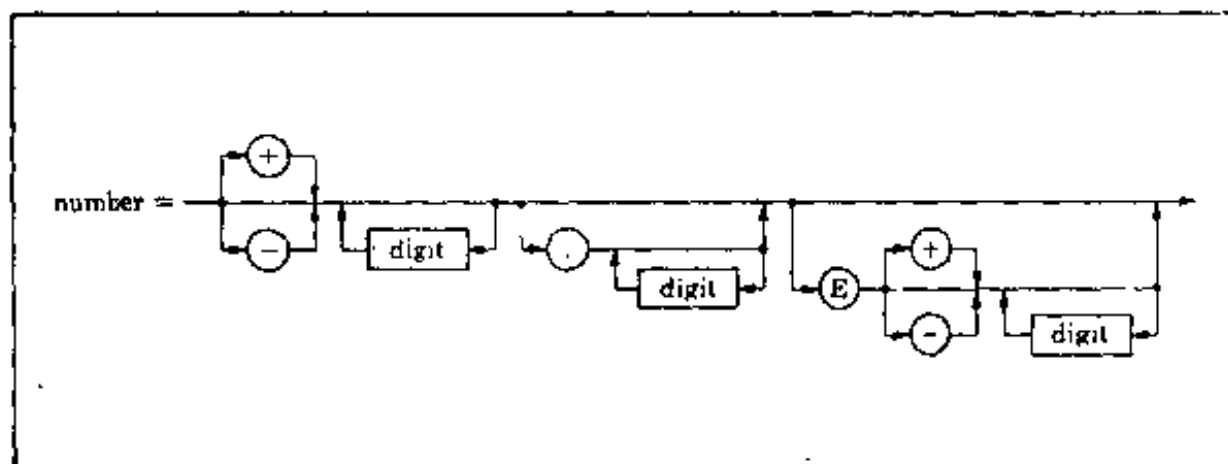


圖 4-6 數的語法圖

注意數的每部份都可任選——甚至於一個空字串（`null string`）都是合法的數。

由複雜的語法圖可知 `stox` 也是很複雜的，希將其大綱，以簡單的虛擬代碼寫好：

```
begin
  get optional leading sign
  get (zero or more) digits
  if decimal point is seen
    get digits after decimal point, if any
  if "E" is seen
    get exponent
  combine into extended real
end
```

不幸地很，大多數的 `stox` 的邏輯與第三章的 `stof` 程序類似，若小心處理有效數位（`nsig`）和小數點後面的數位（`nafter`），即使很複雜，也能控制的很好。

```
function stox(var s: string; var i: integer; var x: xreal): xresult;
  ( Convert string to extended real )

const
  HUGE = 9999;          ( Limit on converted exponent )

var
  rawexpo, nsig, nafter, epart, nesig: integer;
  negepart, negnum: boolean;
  rawfrac: register;
  c: char;
  f: fixed;

begin ( stox )
  rawfrac := 0;
  nsig := 0;
  nafter := 0;
  epart := 0;
  negnum := FALSE;
  negepart := FALSE;
  if gnbchar(s, i, c) in ['+', '-'] then begin
    negnum := (c = '-');
    i := i + 1;
  end;
  while gnbchar(s, i, c) = '0' do
    i := i + 1;
  while gnbchar(s, i, c) in ['0'..'9'] do begin
    if nsig < REGSIZE - 1 then
      rawfrac := 10 * rawfrac + ord(c) - 48;
    nsig := nsig + 1;
    i := i + 1;
  end;
  if c = '.' then begin
    i := i + 1;
    if nsig = 0 then
```


• 4 打碎數字：一般目的計算器 •

```

while gnbchar(s, i, c) = '0' do begin
  nafter := nafter + 1;
  i := i + 1
end;
while gnbchar(s, i, c) in ['0'..'9'] do begin
  if nsig < REGSIZE - 1 then
    rawfrac := 10 * rawfrac + ord(c) - 48;
  nsig := nsig + 1;
  nafter := nafter + 1;
  i := i + 1
end
end;
if gnbchar(s, i, c) in ['E', 'e'] then begin
  i := i + 1;
  epart := HUGE;
  if stof(s, i, 0, f) then begin
    negepart := (f < 0);
    if negepart then
      f := -f;
    if f < HUGE then
      epart := trunc(f)
  end
end;
if (nsig < REGSIZE - 1) then
  rawfrac := rawfrac * ten[REGSIZE - 1 - nsig];
if negnum then
  rawfrac := -rawfrac;
if negepart then
  rawexpo := nsig - nafter + EXCESS - epart
else
  rawexpo := nsig - nafter + EXCESS + epart;
stox := xnorm(rawexpo, rawfrac, x)
end;

```

令人感到驚異的是，**stox** 函數最困難的部份，居然是處理使用者鍵入一個龐大的指數。**stox** 用粗暴的方式解決：首先把指數部份 **epart** 定一個很大的初值 **HUGE** 為 9999，並且利用 **stof** 去累計指數。如果 **stof** 轉換該指數失敗（回轉 **FALSE**），或指數大於 **HUGE**，**epart** 即回轉值為 **HUGE**；否則 **epart** 即被指定一個值，該值是你在螢幕上看到的值。有錯誤的話，**xnorm** 常式即回轉一個適當的錯誤代碼。

將廣域實數轉換成一字串

(Converting an Extended Real To a String)

xtos 常式將廣域實數（extended real）轉換成一字串。它根據實數的格式（定點或科學格式）來決定呼叫適當的常式。

```
function xtos(x: xreal; fmt: format; ndigs: integer; var s: string): xresult;
( Convert extended real to string )

(-----)
( Modules to be inserted here: )
(      xtosci                     )
(      xtofix                     )
(-----)

begin { xtos }
  case fmt of
    FIXEDPOINT:
      xtos := xtofix(x, ndigs, s);
    SCIENTIFIC:
      xtos := xtosci(x, ndigs, s)
  end
end;
```

因為輸入 **xreal** 時採捨入的方法，可能造成溢位，**xtos** 必須回轉一個 **xresult** 值，告訴呼叫者，**xtos** 的翻譯工作順利。

定點格式由 **xtofix** 常式完成之，其虛擬代碼為：

```
begin
  if number is too large to be presented compactly
    convert to scientific notation
  else if number is too small to be presented accurately
    convert to scientific notation
  else
    if number has more than ndigs digits after decimal
      round number to ndigs digits
    convert to fixed-point
end
```

所謂捨入是，把一個合理的因素加到數字去，或捨掉若干個數位。例如，我們希望小數點後面有三位數，就加一數 0.0005。加上這捨入因素，再把數字的分數部份除以 10，在小數點後面超過 **ndigs** 數位的部份捨掉。再利用前章的 **ftos** 常式，把該結果轉換成字串。

這個轉化寫成 PASCAL 是：

• 4 打碎數字：一般目的計算器 •

```
function xtofix(x: xreal; ndigs: integer; var s: string): xresult;
( Convert extended real to fixed point format )

var
  negnum: boolean;
  result: xresult;
  round: xreal;

begin ( xtofix )
  if x.expo - EXCESS > FIXSIZE - ndigs then ( too big )
    result := xtosci(x, FIXSIZE - 1, s)
  else if (x.expo - EXCESS < 1 - ndigs) and (x.frac <> 0) then ( too small )
    result := xtosci(x, FIXSIZE - 1, s)
  else begin
    result := OK;
    if x.expo - EXCESS < FIXSIZE - ndigs then begin ( round )
      negnum := x.frac < 0;
      if negnum then
        x.frac := -x.frac;
      round.frac := 5 * ten[FIXSIZE - 1];
      round.expo := EXCESS - ndigs;
      result := xadd(x, round, x);
      if negnum then
        x.frac := -x.frac;
    end;
    if result = OK then
      fto(x.frac div ten[FIXSIZE - ndigs - x.expo + EXCESS], 0, ndigs, s)
    end;
    xtofix := result
  end;
end;
```

至於 **xtosci** 常式，是用來處理把一數轉換成科學記號，其方法與上述類似，虛擬代碼是：

```
begin
  if fewer than FIXSIZE digits are desired in answer and number is non-zero
    round number to ndigs digits
  convert frac to string
  convert expo to string
  combine into final result
end
```

ftos 常式實行「數字轉換字串」的單調事情，寫成 PASCAL 是：

```
function xtosci(x: xreal; ndigs: integer; var s: string): xresult;
( Convert extended real to scientific notation )
```

```

var
  negnum: boolean;
  result: xresult;
  rawfrac: register;
  rawexpo: integer;
  s1, s2: string;

begin ( xtosci )
  result := OK;
  if (ndigs < FIXSIZE - 1) and (x.frac <> 0) then begin ( round )
    negnum := (x.frac < 0);
    if negnum then
      x.frac := -x.frac;
    rawfrac := x.frac * ten[FIXSIZE + 1] + 5 * ten[2 * FIXSIZE - ndigs - 1];
    rawexpo := x.expo;
    result := xnorm(rawexpo, rawfrac, x);
    if negnum then
      x.frac := -x.frac;
  end;
  if result = OK then begin
    ftos(x.frac div ten[FIXSIZE - ndigs - 1], 0, ndigs, s1);
    if x.frac = 0 then
      s2 := '0'
    else
      ftos(x.expo - EXCESS - 1, 0, 0, s2);
    s := concat(s1, 'e', s2);
  end;
  xtosci := result;
end;

```

變數的儲存和檢索 (Variable Storage and Retrieval)

剩下最簡的任務是：儲存和檢索變數。evalstmt 呼叫 assign，將 x 指派給識別符號 id。

```

procedure assign(id: identifier; x: xreal);
( Assign value to identifier )

{-----}
{ Modules to be inserted here: }
{      insertnode      }
{-----}

begin ( assign )
  insertnode(root, id, x);
end;

```

assign 呼叫 insertnode，以便去找包含變數 id 的節。如果在樹中無法找到 id，insertnode 就在樹上新加一個節，該節包含 id 和 id 的值 x。如果找到了，insertnode 只將 id 的節的值改為 x。

• 4 打碎數字：一般目的計算器 •

因二分樹天生就是一遞迴資料結構，**insertnode** 常式也是遞迴。

```
procedure insertnode(var p: nodeptr; var id: identifier; var x: xreal);
{ Add identifier to tree or change value of existing identifier }

begin ( insertnode )
  if p = NIL then begin { id not in tree, add it by creating new node }
    new(p);
    p^.name := id;
    p^.value := x;
    p^.left := NIL;
    p^.right := NIL;
  end
  else if id < p^.name then { search left subtree }
    insertnode(p^.left, id, x)
  else if id > p^.name then { search right subtree }
    insertnode(p^.right, id, x)
  else { Identifier found, change old value }
    p^.value := x;
end;
```

許多人覺得遞迴常式非常美，也有很多人覺得混亂，如你覺得混亂，請仔細讀下一段。

用英文來描述 **insertnode** 的邏輯工作如下：若節的指標 **p**，把 **NIL** 傳送給 **insertnode**，即表示搜尋失敗。利用標準 **PASCAL** 的 **new** 程序，建立一新節，指標為 **p**，並指派新節的欄位給識別符號的名字和值。同時也設定新節的子樹指標（**subtree pointers**）**left** 和 **right** 為 **NIL**。（注意，因為 **p** 是一個 **var** 參數；它的新值是用來回轉給呼叫常式。如果它並不同轉，會發生什麼事呢？要記住二分樹的根（**root**）已定初值 **NIL**。）

如果傳給 **insertnode** 的指標不是 **NIL**，就檢查它指著什麼。如果 **id** 小於節的識別欄（**p^.name**），就必須搜尋左子樹（由 **p^.left** 指向）。這個搜尋只需呼叫 **insertnode** 遞迴常式，即能完成。同理，如 **id** 大於 **p^.name**，呼叫 **insertnode** 自右邊子樹開始搜尋。

若 **id** 既大大於 **p^.name**，則 **id** 必然等於 **p^.name**

。我們必然會發現，該節已加到樹上了，這時，該節的 **value** 欄，設定新值 **x**。

上述解釋如何將一變數，放到樹上，至於檢索變數值是否由 **evalfact** 提出需求呢？這件工作由函數 **evalid** 處理之：

```
function evalid(id: identifier; var x: xreal): xresult;
{ Get value of identifier }

var
  p: nodeptr;

{-----}
{ Modules to be inserted here: }
{      searchtree      }
{-----}

begin { evalid }
  p := searchtree(root, id);
  if p = NIL then begin { not found }
    remark(concat('Undefined identifier: ', id));
    x.expo := MINEXP;
    x.frac := 0
  end
  else
    x := p^.value;
  evalid := OK
end;
```

evalid 呼叫 **searchtree** 在樹結構尋找 **id**；如果該識別符號找不到，**searchtree** 即回轉 **NIL**。**evalid** 即通知使用人，這變數未定義，並回轉 **0** 作為該變數之值。

如果該識別符號在樹結構，**searchtree** 回轉一個指標，指向它的結，並且將該節的值欄當作識別符號的值：

```
function searchtree(p: nodeptr; var id: identifier): nodeptr;
{ Search tree for identifier }

begin { searchtree }
  if p = NIL then { not in tree }
    searchtree := NIL
  else if id < p^.name then { search left subtree }
    searchtree := searchtree(p^.left, id)
  else if id > p^.name then { search right subtree }
    searchtree := searchtree(p^.right, id)
  else { found it }
    searchtree := p
end;
```

就如同所期望的一樣，**searchtree** 函數的邏輯與 **insertnode** 相似。

注意，**insertnode** 和 **searchtree** 兩函數的設計，建築在 **APPLE** 和 **UCSD PASCAL** 之上，任意長的字串可以直接用相關運算（**>**，**<**，**>=**，**<=**，**<>**，和**=**）來比較，在標準 **PASCAL**，就只允許兩字串（**packed array of char**）在宣告時定義等長，才能用相關運算比較之。

建議（**Suggestions**）

改進 **calc** 的方法很多，茲討論如下：

效率（**Efficiency**）

無庸置疑地，**calc** 並非很快的計算工具，雖然它能滿足大部份人的需要。你決定改進 **calc** 的效率嗎。要決定那個常式最花時間，此為考慮最佳化的要件。有時僅憑一般性的知識和以前的經驗，即能解決。其他時候需要直接衡量。

有些版本的 **PASCAL** 提供一個「自動計算常式被呼叫的次數」的功能，這功能可由使用者任意選擇，也有助於發現那些常式須要最佳化。如你的版本，不具這個特性，可增加一些敘述來遂行之；例如，一個「粗暴強迫」的方法是鍵入常式時，將其名字排印。或由程式取得一個內部陣列，每衡量一個常式，就用到一個元素。每一個常式在登錄時，其於陣列對應的元素就增加 1。

如你的系統有一即時鐘（**real-time clock**），就可以累積計算各個常式花費的時間。

每做一次修正，就衡量出速度改進的結果，如此便能知道那一個方法最能增加效率。

錯誤復原（**Error Recovery**）

如前面所述，`calc` 僅掃描輸入字串，盡快將其數值結果報告。你可藉之改進 `calc`，處理龐大錯誤和無意義的輸入。例如，如果 `calc` 在結束之前停止評估敘述，即提出警告，並指示字串中停止求值的位置。

可信度 (Reliability)

雖然 `calc` 並不受溢位或其他計算方面的影響，它的問題在於記憶體如何處理這些麻煩。每個變數被定義時都需要占用記憶體。程序和函數每次被呼叫的時候，也需用占用大量記憶體（離開副常式時，記憶就自由了）。如果變數太多，求值的表示太複雜，都會造成「記憶不足」的錯誤。處理這些狀況，對於程式設計師而言，是很大的挑戰。

增加的特性 (Adding Features)

`calc` 最明顯的缺點是缺少一些功能。對於大多數的程式語言，表示

`SIN (x)`

必須求 `x` 的正弦值。並於其他三角函數、演算法、指數、平方根、等等都可以增加。`calc` 也可使用指數運算，所以可做下列計算

$$\begin{array}{l} 2^3 = 8.00 \\ \text{[S2]} \quad 8^8 = 1.68e7 \end{array}$$

第一步驟，當然是決定在語法圖內何處出現，新的功能或運算，其次決定代碼的部份作些修訂，增加新的特性，不要忘了，增加新的錯誤代碼尚未被考慮到（例如，求負數平方根）。

其他特性也必然有用，可以調整的數基（`radix`）對設計師也有幫助，可利用它們以 16 進位形式表達結果，也可用 8

• 4 打碎數字：一般目的計算器 •

進位，或其他基數處理之。此外，尚可選擇複雜的計算。

增加的命令，可以促使 `calc` 顯示所有流程識別符號；增加一個輸送好的輸入格式，和結果到指標去，使得 `calc` 與排印計算器類似；增加一個修訂編輯的能力，讓使用人重新計算那僅做些許改變的公式，並增加將現行已定義的變數載入或儲存到檔案內。

`calc` 尚可增加一些功能，由使用人指明重複計算，改變決定，等”。也可以將這些功能，自磁碟載入、或儲 `calc` 程式。

推薦閱讀 (Recommended Reading)

電子計算機的算術運算請參考 *Seminumerical Algorithms* (D.Knuth 著) 的所有算術的常式皆本於此書。

語言分析方面的一般主題，見 N.wirth 著的 *Algorithms + Data structures = programs* 第五章。

大部份用 PASCAL 寫的書都提到二分樹。A.Tenenbaum 和 M.Augestein 著 *Data structures Using PASCAL* 包括許多二分樹和其他通俗的資料結構。P.Grogono 和 Wirth 的 *Programming in PASCAL* 也討論二分樹結構。

5

本文檔案工具

(*Text File Tools*)

本章內將發展若干常式，來操作本文檔案：程式、文件、或其他人類可讀的資訊儲存在電子計算機裡，同時也建立 **print** 公用程式排印本文檔。

多數的設計師，甚至於電子計算機使用人，至少有一個直覺構想，本文檔是一個「字元被重新安排成一行字串，其長度不定」的檔案。它也是被電子計算機用來儲存資訊，可供人類閱讀的一種方法。於是乎，很多用戶從事本文檔方面的設計：使用編譯器（**editors**）建立或修正本文檔，編譯器將程式檔編譯成電子計算機認識的代碼，本文格式器重新安排本文檔，使之更令眼睛滿意的格式，等等。

因為本文檔經常使用到，所以是值得投資人力，去設計改良的工具。又受限於 **PASCAL** 語言、和特殊版本的 **PASCAL**，強迫我們對於在第一章內不同的「好」的關鍵，作一取捨。

選替 (*Trade-Offs*)

一套選替於標準 **PASCAL** 內建輸入／出程式（不是用於交作系統）。正如前面提到的，標準 **PASCAL** 並未提供一個

有效方法，供其在程式內根據名字接達一個檔案；例如，它阻止我們寫一個程式，允許使用人說明一個交作式輸入檔。更大的缺點是，標準 P A S C A L 進行讀或寫檔案時，並未提供偵錯的方法；這妨碍易傳性的代碼，讓使用人在遇到這類錯誤時，無法復原。

許多版本的 P A S C A L 均提供解決之道，但是它們的方法皆非易傳性。這就是一個選替：爲了改良程式的彈性和可信度，不得不將一些易傳性犧牲掉了。

第二是效率的選替。A P P L E P A S C A L 的本文檔，利用內建程序 **read** 和 **write** 處理輸入／出，不可能慢。要增加效率，可能是利用非易傳性的常式來讀整塊本文，而非單一或一行字串，甚而利用組合語言改寫 P A S C A L 常式。於是在增加效率和減低易傳性之間，就有一個選替。

利用非易傳性的語言特性，和程式寫作技巧，導致程式碼，不如單純使用內建程序 **read** 和 **write** 常式清楚。

我們選擇了增大彈性、信賴度、和效率，却降低了易傳性和擴充性，以致不夠清楚。

限制我們的工具，使用少量的常式來改良攜帶性的問題，這對於各種環境，都是很容易改寫的。表 5-1 列舉了本文檔案工具的表列與各個常式的資訊。

表 5-1 本文檔案工具

Module	Purpose
tlopen	Open an existing text file for input
tfcreate	Create a new text file for output
tfread	Read a string from a text file
tfwrite	Write a string to a text file
tfclose	Close a previously opened or created text file
gettfname	Obtain the name of a text file from the user

APPLE PASCAL本文檔格式 (Apple Pascal Text File Format)

當我們使用標準PASCAL的read和write常式，作為本文檔的I/O，就不必擔心在本文檔中使用局部作業系統（local operating system）放置特別的格式。很不幸，我們決定增強效率，就必須學習APPLE PASCAL作業系統的本文檔處理的詳細方法。

- APPLE PASCAL磁碟檔由512位元組塊組成，本文檔在這種檔內是特別的案例。每一個本文檔由1024——位元組頁（1024 - Byte pages），每一頁包括2塊。
- 檔案的第一頁並無本文；它包含資料為系統編譯器使用之。APPLE PASCAL的內建本文檔常式自動地處理標題頁。
- 本文檔的每一順序頁包括真正的本文。每一行本文不會分印於兩頁；每頁行數是一整數。每頁最後剩餘的空間附加<NUL>（ASCII 0）字元。
- 列印的每一行結尾註記<CR>（ASCII 13）字元。
- 每行的開頭空白可任選由下列方式解碼：各行第一字元是一個<DLE>（ASCII 16）；第一位元組包括引導空白字元的個數加32（模256）。引導空白隱藏的基本目的在於節省磁碟空間。

本文檔資料結構 (Text File Data Structure)

設計本文檔案工具，最基本的是要考慮彈性，信賴度，和效率。為了確保做到信賴度，所有的常式，在遇到錯誤時，皆

回轉幾種錯誤代碼，以利錯誤復原。爲了有效率起見，APPLE PASCAL 的 **blockread** 和 **blockwrite** 常式用來自磁碟本文檔讀寫一頁。

本文檔案常式定義下列資料型態：

```
<const>
  GOODIO = 0;
  ENDFILE = -1;
  PAGESIZE = 1024;
  MAXFNAME = 23;

<type>
  iostatus = integer;
  tfile = file;
  page = packed array [0..PAGESIZE] of char;
  tfrec = record
    buf: page;
    bufptr: integer;
    mode: (RDMODE, WRMODE);
  end;
  filename = string[MAXFNAME];
```

我們的常式，以變數 **type iostatus** 回轉錯誤代碼，0（用助憶符號 **GOODIO** 表之）代表無誤，-1（助憶符號 **ENDFILE**）表示輸入資料全部讀完。其他值均表示有錯，**iostatus** 定義爲：

```
<type>
  iostatus = (GOODIO, ENDFILE, ERR);
```

這個方法消除常數宣告的 **GOODIO** 和 **ENDFILE**，同時也會喪失一些錯誤代碼方面的資訊。在我們的設計之中，不同的錯誤就有不同的錯誤代碼，所以呼叫程式就有不同的動作。

blockread 和 **blockwrite** 需要我們使用一些特別的 APPLE PASCAL 檔案型態，這檔案型態即所謂的 **untyped** 檔案。

• 5 本文檔案工具 •

使用 **tfrec** 記錄型態保存既知的本文檔案。該記錄包括一個 1025——位元組緩衝器 (**buf**)，掌握記憶中檔案的流程頁。只有緩衝器內的第一個 1024 位元組，是真正的被讀或寫。第 1025 位元組 (1024 元素) 被當作崗哨，表示本頁的終點。崗哨位元組永遠包含一個 **<NUL>**，它保證每頁的尾端最少有一個 **<NUL>**。一個整數 (變數 **bufptr**) 掌握下一行要讀或寫的本文，在緩衝器的位置。變數 **mod** 表示檔案正是被讀 (**RDMODE**) 或是被寫 (**WRMODE**)。

如我們已能夠把一所予的檔案保存在記錄內，則資料結構就會很清楚。

```
<type>
  tfrec = record
    f: file;
    buf: page;
    bufptr: integer;
    mode: iomode
  end;
```

然而，我們無法這麼做，因為 **APPLE PASCAL** 不允許記錄內包含檔案型態。檔案陣列和指標指向檔案，也是不合法的。

最後，檔案名字當作字串儲存，字串長度最長為 23 個字元，此亦為 **APPLE PASCAL** 中作業系統的限制上限。

打開輸入檔常式 (**tfopen**)

下列常式為輸入資料打開一已定名的本文檔：

```
function tfopen(var f: tfile; var tf: tfrec; name: filename;
                var status: iostatus): iostatus;
  ( Open text file for input )
var
```

```
    nbr: integer;
begin ( tfopen )
  ($i-)
  reset(f, name);
  status := ioreult;
  if status = GOODIO then
    with tf do begin
      nbr := blockread(f, buf, 2, 2);
      status := ioreult;
      if (status = GOODIO) and (nbr <> 2) then
        status := ENDFILE;
      buf[PAGESIZE] := chr(0);
      bufptr := 0;
      mode := RDMODE
    end;
  tfopen := status
  ($i+)
end;
```

我們討論 APPLE PASCAL 內建常式 **reset** 和 **ioreult** 與第三章內討論的編譯指引 { \$ i + } 和 { \$ i - } 一樣。

此外，**tfopen** 利用 APPLE PASCAL 的內建 **blockread** 函數，自檔案讀進第一頁的本文。（注意，要記住檔案的第一個二塊並不合任何本文）。呼叫

```
    nbr := blockread(f, buf, i, j)
```

自 untyped 檔案 **f** 讀第 **j** 塊起 **i** 塊到緩衝陣列 **buf** 去。（檔案中的第一塊被認為塊號 0。）而 **buf** 參數可能用來表示陣列起點位置。起始塊參數 **j** 為選擇性的；如果不選 **j** 表檔案的下一塊已被讀。本常式回轉塊的個數當作函數結果。當讀了最後一塊，函數 **eof (f)** 變成 **TRUE**；否則它就成了 **FALSE**。

tfopen 讀進記憶體一頁時，設定緩衝指示器指向該頁的第一個字元。正如前述，緩衝器內的最後一個字元，就像一崗哨；**tfopen** 將之設定為 **<NUL>**。最後，本常式設定 **mode** 為 **RDMODE**，表示我正從檔案讀本文。

如果在 **reset** 或 **blockread** 運作時，發生錯誤，**tfopen** 即從 **ioreult** 那取得錯誤代碼，將之回轉給呼叫常式，皆以 **var** 參數 **status** 做為函數的結果。如果，基於某些理由，從

ioresult 那並未取得錯誤代碼，但 **blockread** 並未如預期一樣讀二塊，**tfopen** 回轉一個 **ENDFILE** 錯誤代碼給呼叫常式。

建立新檔常式 (**tfcreate**)

```
function tfcreate(var f: tfile; var tf: tfile; name: filename;
                  var status: iostatus): iostatus;
  ( Create new text file for output )

var
  nbw: integer;
begin ( tfcreate )
  ($i-)
  rewrite(f, name);
  status := ioresult;
  if status = GOODIO then
    with tf do begin
      fillchar(buf, PAGE_SIZE, chr(0));
      nbw := blockwrite(f, buf, 2);
      status := ioresult;
      if (status = GOODIO) and (nbw <> 2) then
        status := 8; ( No room )
        mode := WRMODE;
        bufptr := 0
      end;
      tfcreate := status
    end;
  ($i+)
end;
```

tfcreat 爲了要與 **APPLE PASCAL** 的本文檔格式相容，它必須寫二塊本文檔，我們選擇用 **<NUL>** 字元，很簡單就解決了問題。

內建常式 **rewrite (f , name)** 建立一新檔，即所謂與檔案變數有關的 **name**，如該 **name** 不合法，或不能接達，就產生一錯誤代碼。

內建常式 **blockwrite** 與 **blockread** 相反，呼叫

nbw := blockwrite(f, buf, i, j)

從 **buf** 陣列的第 **j** 塊起寫 **i** 塊到 **untyped** 檔案 **f**。而 **buf** 參數可做爲下標 (**subscripted**)，以指示陣列的起始位置。如果

沒有 **j** 參數，即將檔案循序寫進下一塊。被寫的塊數當作函數結果。

同時利用內建常式 **fillchar**，很快的用 **<NUL>** 字元，將 **buf** 陣列填滿。呼叫

fillchar(buf, i, c)

把 **buf** 陣列的 **i** 個位元組，指派給字元 **c**。**buf** 參數也許做下標，表示陣列開始位置。**for loop** 是較具易傳性，然而 **fillchar** 比較快。

當 **var** 參數 **status** 和函數結果有錯，**tfcrcate** 回轉錯誤代碼。如果 **ioresalt** 並不產生 **I/O** 的錯誤，而 **blockwrite** 告訴我們，並未將兩塊寫到檔案，**tfcrcate** 即回轉錯誤代碼 8，此即 **APPLE/UCSD PASCAL** 的錯誤代碼，表示磁碟空間不夠。

關閉檔案 (**tfclose**)

當程式使用完畢一個本文檔，即呼叫 **tfclose** 來關閉它：

```
function tfclose(var f: tfile; var tf: tfrec; var status: iostatus): iostatus;
(* Close text file *)

var
    nbw: integer;

begin (* tfclose *)
    (*S1-*)
    status := GOODIO;
    with tf do
        if mode = WRMODE then begin
            if bufptr > 0 then begin
                fillchar(buf[bufptr], PAGESIZE - bufptr, chr(0));
                nbw := blockwrite(f, buf, 2);
                status := ioresult
            end;
            close(f, LOCK);
            if status = GOODIO then
                status := ioresult
        end
    end
end
```

```
    else begin { Read mode }  
        close(f);  
        status := ioreult  
    end;  
    tfclose := status  
    ($!+)  
end;
```

檔案正被寫的時候，**tfclose** 核對在記憶裡的頁次，是否還有未被寫到磁碟的任何文字列，如果是的，最後一頁就附加一些 **<NUL>** 字元，一同寫到磁碟上。在任意情況下，該檔利用 **APPLE PASCAL** 內建程序，把第二參數集合一同設定 **Lock**，並把檔案關閉。

如果正在處理的是讀檔，呼叫 **close** 檔時，並不含第二參數。

無論是何種情形，錯誤代碼被當做 **var** 參數 **status** 和函數結果，被回轉。

自本文檔讀一字串 **tfread**

tfread 常式自本文檔讀下一列，到一字串變數，該常式必須檢查正常核錯之外的兩個特殊狀況。

第一種狀況是 **end-of-file**，解決之道很簡單。讀進檔案的最後一行時，循序呼叫 **tfread**，並回轉 **ENDFILE** 取代 **GOODIO**，作為 **I/O** 狀態。

第二種狀況並不明顯，却很重要。本文檔的行此字串變數大會發生什麼情況？我們並不希望 **tfread** 把那超出字串的最大長度的部份視同無意，也不寄望 **tfread** 回復「太長的行作為二個（以上）的字串，而不給呼叫程式任何信號」。

解決方法是：正常地自頁緩衝器附加字元，到字串的尾端，直到字串長度到達最大長度或 **<CR>** **end-of-line** 字元為止。其他字元則藉呼叫 **tfread** 檢索；呼叫的次數，與讀行的

<CR> end-of-line 次數一樣多。

tfread 常式如下：

```
function tfclose(var f: tfile; var tf: tfile; var status: iostatus): iostatus;
( Close text file )

var
  nbw: integer;

begin ( tfclose )
  ($i-)
  status := GOODIO;
  with tf do
    if mode = WRMODE then begin
      if bufptr > 0 then begin
        fillchar(buf[bufptr], PAGE_SIZE - bufptr, chr(0));
        nbw := blockwrite(f, buf, 2);
        status := ioresult
      end;
      close(f, LOCK);
      if status = GOODIO then
        status := ioresult
    end
    else begin ( Read mode )
      close(f);
      status := ioresult
    end;
  end;
  tfclose := status
  ($i+)
end;
```

輸入字串以 **var** 參數 **s** 回轉。該常式呼叫 **tfread** 用 **maxlen** 來說明字串 **s** 可接受最大的字元個數。它允許呼叫程式，根據應用方面的需要，調整輸入字串的能力。和平常一樣，任意錯誤代碼，當作函數結果和 **var** 參數 **status** 被回轉。

tfread 首先呼叫 **grabline**，從 **buf** 轉移下行到字串變數 **s**，通常這是必要的。如果緩衝器是空的，**grabline** 回轉一空的字串（長度為0）。如此，**tfread** 檢查 **eof** 函數，以了解檔案的最後頁是否已讀過了。如 **eof(f)** 是 **TRUE**，表示檔案結束；**tfread** 回轉 **ENDFILE** 給呼叫常式。如 **eof(f)** 是 **FALSE**，最少還有一頁沒讀，於是，**tfread** 呼叫 **block-read**，取得下頁，重定緩衝指示器，最後再呼叫 **grabline**，自新頁取第一行到字串。

最後將用 APPCE II 的 6502 組合語言寫 **grabline** 常式。

首先用 PASCAL 表示。這樣會有若干好處：第一，在其翻譯成組合語言之前，可以設計並偵錯其代碼邏輯。第二，讀者的其他電子計算機，可直接利用 PASCAL 常式，或用 6502 代碼作墊腳石，改成其他機器自己的組合語言的常式。

grabline 的虛擬碼非常複雜。最主要的複雜狀況是把在本文檔內任何隱藏不現的 <DLE> 空白字元，變成真正的空白字元，以便字串回轉。

首下描繪出 **grabline** 概要如下：

```
begin
  set string = '' (null string)
  repeat
    get next character from buffer
    if character is not <NUL>
      append it to string
  until (character is <NUL>) or (character is <CR>) or (max length is reached)
end
```

grabline 將字元的尾端附加字元，直到 <NUL> 字元或 <CR>，或已是最大字串長度為止。

在 <DLE> 被處理時，從緩衝器內取下個字元。爲使 **grabline** 盡可能被可信，必須正確的處理「從未發生」的情況。這種情況可能是(1)在中間或尾端有個 encryption，(2)二個或多個 encryption，或(3)一個 encryption 指示輸出字串有太多的空白字元。以上都能處理時，「取下個字元」運作虛擬代碼如下：

```
get next character from buffer:
  ( Assume bp is current position in buffer, c is returned character )

  c = buf[bp]
  bp = bp + 1
  while c is a <DLE> character
    if buf[bp] is a <SPACE> (no more blanks in encryption)
      c = buf[bp + 1]
      bp = bp + 2 (point bp at next character)
    else (more blanks to decode)
      c = <SPACE>
      decrement buf[bp] (mod 256)
      bp = bp - 1 (point bp back at <DLE> for next time)
  end-while
```

grabline 寫成 PASCAL :

```
procedure grabline(var buf: page; var i: integer; var s: string;
                  maxlen: integer);
{ Get line from page buffer, decode DLE encryption }

var
  len: integer;
  done: boolean;
  c: char;

begin { grabline }
  len := 0;
  s := '';
  repeat
    c := buf[i];
    i := i + 1;
    while c = chr(16) do
      if buf[i] = ' ' then begin
        c := buf[i + 1];
        i := i + 2;
      end
      else begin
        buf[i] := chr((ord(buf[i]) + 255) mod 256);
        i := i - 1;
        c := ' ';
      end;
    done := (c = chr(0));
    if not done then begin
      addchar(s, c, maxlen);
      len := len + 1;
      done := (c = chr(13)) or (len >= maxlen)
    end
  until done
end;
```

在 **grabline** 中將緩衝字元遞減的運作如

```
buf[i] := chr((ord(buf[i]) + 255) mod 256)
```

取代自然的

```
buf[i] := chr((ord(buf[i]) - 1) mod 256)
```

這是因為APPLE PASCAL實行 mod 運算，並非是標準狀態。在APPLE PASCAL中，表式 $(-1) \bmod 256$ 之值為1，而非255。為了避免這個問題，總是小心，並保證引數作mod 運算時不可為負數。

真正程式原始本文裡，**grabline** 很可能是 **tread** 函數內

部的巢狀結構。

將一字串寫到本文檔的常式 (**tfwrite**)

tfwrite 將一字串寫到本文檔，它與 **tread** 常式相反：

```
function tfwrite(var f: tfile; var tf: tfrec; var s: string;
                  var status: iostatus): iostatus;
{ Write string to text file }

var
  nbw: integer;

begin { tfwrite }
  ($i-)
  status := GOODIO;
  crunch(s);
  with tf do begin
    if bufptr + length(s) > PAGE_SIZE then begin
      fillchar(buf[bufptr], PAGE_SIZE - bufptr, chr(0));
      nbw := blockwrite(f, buf, 2);
      status := ioreult;
      if (status = GOODIO) and (nbw <> 2) then
        status := 8; { No room }
      bufptr := 0
    end;
    dropline(buf, bufptr, s)
  end;
  tfwrite := status
  ($i+)
end;
```

tfwrite 首先呼叫 **crunch** 以便將輸出字串中的引導空白字元編碼 (**encode**) 以便將輸出字串中的引導空白字元編碼 (**encode**)，其次再查核流程頁緩衝器是否有足夠空間掌握輸出行；如沒有空間，頁緩衝器即附加 <NUL>，並把該頁寫到輸出檔。緩衝指示器被回轉到緩衝器的開端。**tfwrite** 使用 **dropline** 把字串的字元抄到緩衝器。

crunch 常式如下：

```
procedure dropline(var buf: page; var i: integer; var s: string);
{ Put string in output buffer }

begin { dropline }
  if length(s) > 0 then begin
    moveleft(s[1], buf[i], length(s));
    i := i + length(s)
  end
end;
```

```
begin ( crunch )
  i := 1;
  c := gnbchar(s, i, c);
  if i > 3 then begin
    s[1] := chr(16);
    s[2] := chr((i + 287) mod 256);
    moveleft(s[i], s[3], length(s) - i + 1);
    ($r-3
    s[0] := chr(length(s) - i + 3)
    ($r+)
  end
end;
```

因爲，將字串的引導空白字元編碼的目的，是爲了節省磁碟的儲存空間，只有在引導空白個數超過（或等於）3個時，才這麼做。零或一個引導空白字元的編碼，使得字串變長；編兩個空白碼、保留字串相同的長度。後面將改以組合語言寫 **crunch**。

crunch 使用 APPLE PASCAL 內建常式 **moveleft**，從記憶體在某處，搬移任意個位元組，到記憶體的其他地方。呼叫

moveleft(source, destination, nbytes)

自 **source** 位置搬移 **nbytes** 位元組，到 **destination**。**source** 和 **destination** 這兩變數，可定義成任何型態（除了檔案型態之外），也可用做爲下標，指示起點位置。抄寫的時候，自 **source** 範圍內的第一個位元組起，向尾端抄 **nbytes** 位元組。和 **fillchar** 常式一樣，可以用具易傳性的 **for** 環（**loop**），但是使用 **moveleft** 比較快。

dropline 程序把字串內的字元，搬移到頁緩衝器，並更新（**update**）緩衝指示器，以指向緩衝器內下一個有效位置，最後再將 **dropline** 用組合語言改寫，做法與 **grabline** 和 **crunch** 相同。**dropline** 用 PASCAL 寫形式如下：


```
procedure dropline(var buf: page; var i: integer; var s: string);
< Put string in output buffer >

begin < dropline >
  if length(s) > 0 then begin
    moveleft(s[1], buf[i], length(s));
    i := i + length(s)
  end
end;
```

自使用人那兒取得本文檔案名字 (**gettfname**)

APPLE PASCAL 與其他作業系統一樣，對於合法的檔案名稱，有唯一和不可攜帶的法則。**gettfname** 利用 **getstring** 自使用人那兒，取得一本文檔。

```
procedure gettfname(var name:filename; col, row:integer; default:filename);
< Get text file name from user >

begin < gettfname >
  ($v-)
  getstring(name, MAXFNAME, col, row, default,
    ['!','.',',','~','_'] = ['$', '[', '=', '?'], TRUE);
  ($v+)
  if (length(name) > 0) and (pos('.TEXT', name) = 0) and
    (length(name) <= MAXFNAME - 5) then begin
    name := concat(name, '.TEXT');
    posstr(name, col, row)
  end
end;
```

gettfname 尚且使用一個我們還設討論的常式 **pos**。呼叫函數

i := pos(sub, s)

搜尋字串 **s** (從開始處)，找字串 **sub** 第一次出現的位置。(在 **gettfname** 之中，**pos** 搜尋字串 **TEXT**) 如果 **pos** 成功了，即回回轉字串 **sub** 在 **s** 字串中的索引，如 **sub** 不在 **s** 中，則 **pos** 回轉 0。

gettfname 強迫本文檔名稱依下列規則輸入：限制 **MAXFNAME** 最長 23 個字元；所有的字母一律使用大寫；除了空

間、金錢符號、左括號、符號和問題等字元之外，可排印的字元皆可做為檔案名稱。這些限制在 APPLE PASCAL 的內部轉換、和系統內其他程式都必須遵守。最後正常的本文檔字尾並沒有 **.TEXT** 出現，**gettfname** 就在輸入字串的尾端把它加上。

有興趣的讀者，也許希望修飾 **gettfname**，做更嚴格的查核，這對於使用人鍵入合法名稱益形困難。尤其是現在的版本中，並無查核被鍵入的名稱之內部語法，於是乎，使用人可以鍵入無效的名字，諸如 **WAYTOOLONGVOL : A.TEXT** 或 **N : O : N : S : E : N : S : E .TEXT**。為了防止任何不合法的名稱，你必須考慮使用人輸入時可能犯什麼錯誤。（你會犯那些錯？如果你曾看別人使用交作程式，他們在輸入時，通常犯什麼錯誤？）

組合語言 (Assembly Language)

把 **grabline**，**crunch**，和 **dropline** 翻譯成組合語言的第一個步驟是利用 PASCAL 主程式宣告它們是外部函數（**external function**）或程序。**grabline** 宣告成：

```
procedure grabline(var buf: page; var bufptr: integer; var s: string;  
                   maxlen: integer); external;  
( Get line from input buffer )
```

dropline 宣告方式：

```
procedure dropline(var buf: page; var bufptr: integer; var s: string); external;  
( Put string in output buffer )
```

crunch 宣告方法是：

```
procedure crunch(var s: string); external;  
( Encode leading blanks in string )
```

這些宣告的邏輯配置，可以在常式內部的巢式結構之內。然而 APPLE PASCAL 却不准宣告部份是在外部常式巢狀結構。下一個最佳選擇是在使用的常式之上方宣告這些常式。

我們無法將 PASCAL 內與組合語言混用考慮的太詳細，尤其是，我們不討論使用組合常式去鏈接（link）PASCAL 主程式。（相同的理由，也可描述如何編譯 PASCAL 程式）我們假定有興趣的讀者，已是知之甚詳。

我們也不討論 6502 或其他 cpu 的結構或指令集（instruction set）。再者，我們認為使用人已知道了，或在他們自己的電子計算機系統中發現。

我們討論重點是，APPLE PASCAL 用與組合語言通信的方法；這些資訊容後再敘。

一呼叫程式將參數依次推到組合語言的堆疊（stack）去，這些參數都在副常式的參數表列中宣告。（因此，它們必須以相反次序爆出（pop）。）如果組合語言副常式被宣告成函數，APPLE PASCAL 也就必須把額外的 4 個位元組推到堆疊去，而成爲那些參數的頂；這額外的位元組，必須被宣告是函數的入口。堆疊頂上的二個位元組，放置副常式的回轉位址。

離開時，如果組合語言副常式被宣告爲一函數，函數結果必須放回堆疊。組合語言常式（被宣告成一程序或函數）也必須推回堆疊二個位元組，它們包含副常式的回轉位址，這二位元組必須是堆疊的頂。

呼叫程式將個別的參數傳送到一副常式，與如何在 PASCAL 主程式中的 internal 宣告有關。一個 var 參數永遠用位址傳遞（堆疊中包括變數的第一個位元組位址）。記錄變數、

陣列變數、和字串也都是由位址傳送。其他資料型態（整數、布耳、字元、實數、次範圍、純量型態、指標、集合，和長整數）被宣告成值（非變數）參數、則由值傳送（把它們的值推到堆疊）。

僅將位址或整數傳送到組合的常式；需要 2 個位元組並且其儲存到記憶內是先存低有效位元組、再存有效位元組。在堆疊的頂是低有效位元組；低有效位元組先自堆疊中爆出，最後推到堆疊的。

組合語言原始本文，在組合語言副常式之後，包括選擇的 macro definition（巨組定義）。第一個巨組定義是 pop，它是一個 16 位元值（2 個位元組）。

```

;      .macro pop
;
;      macro to pop 16-bit argument
;
;      pla
;      sta %1
;      pla
;      sta %1+1
;      .endm
```

巨組定義是常見運作的速記定義，不必重複寫一長序列的 6502 指令去實行一普通運算。例如，我們把下列放到組合語言常式：

```
pop return
```

組譯器（assembler）自動地把它翻譯成如下的一序列 6502 指令：

```
pla
sta return
pla
sta return+1
```

我們還為與 pop 相反的運算 push 定義一個巨指令；它將

- 16 - 位元引數推到堆疊：

```
        .macro push
;
;      macro to push 16-bit argument
;
        lda X1+1
        pha
        lda X1
        pha
        .endm
```

巨指令 **inc 16** 在記憶體內將一個 16- 位元值遞增：

```
        .macro inc16
;
;      macro to increment 16-bit argument
;
        inc X1
        bne $01
        inc X1+1
$01
        .endm
```

最後，**dec 16** 巨指令將一個 16- 位元值遞減：

```
        .macro dec16
;
;      macro to decrement 16-bit argument
;
        inc X1
        dec X1
        bne $02
        dec X1+1
$02
        dec X1
        .endm
```

副常式的組合語言代碼，後跟著巨指令定義。**grabline**

常式如下：

```
;
;      .proc grabline,4
;      procedure to move a line from the input page buffer to string
;      declared in Pascal host as:
;      procedure grabline(var buf:page; var offset:integer; var s:string;
;                          maxlen: integer); external;
```

—

• 5 本文檔案工具 •

```

sbc buf          ; subtract buffer address low byte
sta (offset),y   ; store in offset low byte
iny              ; index offset high byte
lda bp+1         ; get buffer pointer high byte
sbc buf+1        ; subtract buffer address high byte
sta (offset),y   ; store in offset high byte
push return      ; push return address
rts              ; and return

```

dropline 的組合語言代碼是：

```

.proc dropline,s
;
; Procedure to append line to output buffer
;
; Declared in Pascal host as:
;
; procedure dropline(var buf: page; var offset: integer; var s: string);
;                                     external;
return .equ 0          ; return address
buf    .equ 2          ; address of buffer start
offset .equ 4          ; address of buffer offset
s      .equ 6          ; address of string
len    .equ 8          ; length of string

pop return      ; store return address
pop s           ; store string address
pop offset      ; store offset address
pop buf         ; store buffer address

ldy #0          ; index length byte of string
lda (s),y       ; get length of string
beq exit        ; if zero, there's nothing to do
sta len         ; store length
incb s          ; point s at first character of string
;
; set buf to point to current buffer position by adding offset
;
clc             ; prepare to add
lda buf         ; get low byte of buffer address
adc (offset),y  ; add low byte of offset
sta buf         ; store in low byte of buffer address
iny             ; index high byte
lda buf+1       ; get high byte of buffer address
adc (offset),y  ; add high byte of offset
sta buf+1       ; store in high byte of buffer address
;
; move string to buffer
;
nxtbyt  ldy #0          ; index first character
        lda (s),y       ; get next byte from string
        sta (buf),y     ; store in buffer
        iny             ; index next character
        cpy len         ; compare with string length
        bne nxtbyt      ; if non equal, back for next character
;
; finish up by updating offset
;
tys          ; move string length to A
clc          ; prepare to add
ldy #0       ; index low byte of buffer offset
adc (offset),y ; add low byte of buffer offset to length

```

```

        sta (offset),y ; store new low byte of buffer offset
        lda #0
        iny            ; index high byte of buffer offset
        adc (offset),y ; add high byte of buffer offset to carry
        sta (offset),y ; store new high byte of buffer offset
exit    push return    ; push return address
        rts            ; and return

```

最後，crunch 代碼：

```

.proc crunch,1
;
; procedure to encode leading blanks in string
;
; declared in Pascal host as:
;
; procedure crunch(var s: string); external;
;
return .equ 0      ; return address
s      .equ 2      ; string address
len     .equ 4      ; string length
nbsaved .equ 5      ; number of bytes saved by crunching
s1      .equ 6      ; temporary pointer

        pop return  ; store return address
        pop s       ; store string address
        ldy #0      ; index string length byte
        lda (s),y   ; get string length
        sta len     ; and store it
;
; find first nonblank character in string
;
        lda #20      ; put a blank in accumulator
inchr   iny          ; index next string character
        beq havebl   ; if y=0, we've wrapped around (255 blanks)
        cpy len      ; if y > length, exit loop
        beq chkbl
        bcs havebl
chkbl   cmp (s),y    ; is character a blank?
        beq inchr    ; yes, go back for next character
;
; y now contains (# of leading blanks) + 1
; subtract 3 to get number of bytes saved by crunching
; don't crunch if y = 1, 2, or 3
;
havebl   dey
        beq done
        dey
        beq done
        dey
        beq done
        sty nbsaved ; store nbsaved
;
; calculate new length = old length - nbsaved
;
        sec          ; prepare to subtract
        lda len      ; get old length
        sbc nbsaved  ; subtract nbsaved
        ldy #0       ; index length byte
        sta (s),y    ; store new length in string
        sta len      ; and in zero page
;

```



```

;      store blank encryption in string
;      s[1] := <DLE>
;      s[2] := 32 + (# of leading blanks) * 34 + nbsaved
;
      iny          ; index first character
      lda #10      ; get a <DLE>
      sta (s),y    ; store it in string
      iny          ; index second character
      clc          ; prepare to add
      lda nbsaved  ; get nbsaved
      adc #22      ; add 34 (decimal)
      sta (s),y    ; store in string
;
;      move characters in string left
;
      lda s        ; get string address low byte
      clc          ; prepare to add
      adc nbsaved  ; add nbsaved
      sta s1       ; store temp pointer low byte
      lda s+1      ; get string address high byte
      adc #0       ; add carry, if any
      sta s1+1     ; store temp pointer high byte
outch  iny          ; index next character in string
      cpy len      ; if y > new length, we're done
      beq movech
      bcs done
movech lda (s1),y   ; get character from string
      sta (s),y    ; store at new position
      jmp outch    ; back for next character
done   push return  ; push return address
      rts          ; and return

```

後面將針對這個（自 PASCAL 翻成的組合語言）改進，加以評估。

單元和館 (Units and Libraries)

注意程式像 **calc** 和 **goslog**，使用大量的程式設計師定義的資料型態和模組，很快地使程式的代碼變大，又難於處理。後面的程式中，使用 APPLE 和 USSD PASCAL 的兩個特殊性來減緩問題：把相關的程序、函數、資料型態，和常數宣告組合起來，成為分離的編譯單元，並把這些組合放到館內。

我們不去深入探討這種單元，館的結構，以及程式鏈接的方法。

一個單元由兩部份組成：介面部份 (interface section)

和實施部份 (implement section) 。介面部份包括 **const** , **type** , 和 **var** 宣告和標題宣告。實施部份包含真正的代碼和在介面部份為宣告的局部變數。實施部份有時還包含程序和函數自身；就如同其自己的常數，型態，和變數宣告一樣。

根據這些，假定最有用的常式和宣告，已放進單元內，而這些單元則為館所收集，詳見表 5-2 。

一個程式使用一單元，可以參考該單元內的介面部份之內所宣告的項目。實施部份內的被宣告項目，是單元「私有的」

表 5-2 館所收集的單位

Unit Name	Contents (interface)
crtstuff	constant declarations: MAXSTR, MAXCRTCOL, MAXCRTROW type declarations: crtcominand, charset modules: crt, posstr, getkey, addchar, chopchar, getstring, eraseline, center, wait, remark, disptitle, getboolean, ask, gchar, gnbchar
fixstuff	constant declarations: FIXSIZE type declarations: fixed modules: stof, ftoa, getfixed
datestuff	type declarations: date modules: dtoa, getdate
textstuff	constant declarations: PAGESIZE, GOODIO, ENDFILE, MAXFNAME type declarations: lostatus, page, tfile, filename, tfrec modules: tlopen, tcreate, tload, tfwrite, tfclose, gettfname

，並不能被程式接達。例如，表 5-2 內的 **datestuff** 單元，包括 **stod** 程序（把字串轉換成一個 **date**），而 **stod** 在實施部份內被 **getdate** 呼叫。但是 **stod** 並不能為程式呼叫 **datestuff** 而被接達。同理，**textstuff** 單元的實施部份，包括 **grabline**，**crunch**，和 **dropline** 等常式的宣告，但是這些程式並不能由應用程式直接使用。

程式利用 **uses** 宣告單元和館，其在程式敘述是

```
uses
  ($u APPLE2:TOOLSTUFF.CODE) crtstuff, textstuff;
```

注意 PASCAL 編譯器需要呼叫常式，或使用由 **crtstuff** 和 **textstuff** 單元內提供的型態。而且這些單元可以在館檔案 **APPLE 2 : TOOLSTUFF.CODE** 內找到。

館和單元有下列好處：程式設計師可以單獨在某地方使用這些常式，而不分散由好幾個應用程式使用。單元內的代碼，並不因修改程式而重新編譯，所以能節省時間。單元能提供更高的模組化，這遠非程序和函數單獨被使用所能比擬的。如果每一模組被看做一工具，則單元就視同工具箱：相關工具的集合。

最後注意，館和單元都是可選擇的。如你使用的版本並無

度量 (Measurement)

使用 **copy** 常式測試本文檔案，**copy** 將一輸入檔案抄到輸出檔案，我們檢查二者，看看有何不同。**copy** 的主常式是：

```

program copy;
{ Copy text files - benchmark program }

uses
  ($u apple2;toolstuff.code) crtstuff, textstuff;

var
  status: iostatus;
  margin: integer;
  inname, outname: filename;
  infile, outfile: tfile;
  intf, outtf: tfrec;
  done: boolean;
  s: string;

begin { copy }
  margin := (MAXCRTCOL - 35) div 2;
  crt(CLEAR);
  disptitle('copy');
  posstr(' Input file:', margin, 4);
  posstr('Output file:', margin, 6);
  repeat
    gettfname(inname, margin + 12, 4, '');
    gettfname(outname, margin + 12, 6, '');
    remark('Ready to start timing');
    if tfopen(infile, intf, inname, status) = GOODIO then begin
      if tfcreate(outfile, outtf, outname, status) = GOODIO then begin
        repeat
          if tfread(infile, intf, s, MAXSTR, status) = GOODIO then begin
            if tfwrite(outfile, outtf, s, status) <> GOODIO then
              remark('Error writing output file')
            end
          else if status <> ENDFILE then
            remark('Error reading input file')
          until status <> GOODIO;
          if tfclose(outfile, outtf, status) <> GOODIO then
            remark('Error closing output file')
          end
        else
          remark('Error creating output file');
          if tfclose(infile, intf, status) <> GOODIO then
            remark('Error closing input file')
          end
        else
          remark('Error opening input file');
          remark('Stop timing');
          done := not ask('Any more?(Y/N):', MAXCRTROW - 1, FALSE, FALSE);
          eraseline(MAXCRTROW - 1)
        until done;
        crt(CLEAR)
      end.

```

爲了度量不同的設計，在效率上的效果，使用三種不同的 **copy** 自磁碟將二個不同的本文檔抄到另一磁碟上。第一個本文檔是一個 512 行的 PASCAL 程式，內容有 10,722 個字元；第二是手稿檔案，有 488 行、16975 個字元。（也有程式段和本文格式命令。）度量按 <RETURN> 鍵時反應 Ready

to start timing 和出現 stop timing 之間的時間。

第一種 **copy** 稱為 **copy 1**，它使用「標準的」本文檔。用到內建 **text** 檔案型態；**tread** 只呼叫 **readln** 即自輸入檔案讀入，和 **tfwrite** 呼叫 **writeln** 將一行行寫到輸出檔案，錯誤核示如前述。

copy 2 是前面剛寫的 **copy** 程式，使用到 **grabline**，**dropline**，和 **crunch**。**copy 3** 是用組合語言寫的。

這三種不同的 **copy** 程式所執行的本文檔都在表 5-3 列出。我們決定用 PASCAL 處理輸入／出，並用組合語言寫這三個常式；如此其速度增快 10 倍。

表 5-3 Copy 程式之度量

	Program (10,722 characters)	Manuscript (16,975 characters)
copy1	273 seconds	410 seconds
copy2	103 seconds	139 seconds
copy3	26 seconds	37 seconds

copy 最後需要注意的是，詳敘輸入／出常式的程式並非每天都用到。一個真正的「檔案抄寫」常式，一次讀許多檔案塊，再將它們寫到輸出檔案。「檔案抄寫」常式並不會將輸入檔案打破各行、解碼、和再編碼，重組這些行後輸出。此外「檔案抄寫」常式如具彈性，即能夠處理檔案的所有型態。

排印 (print)

print 是一個公用程式，用來排印本文檔。它將排印的輸出檔分成若干頁；每頁都印出頁次、檔案名字、和流程日期。

開始排印時，它詢問日期。鍵入日期之後，程式在螢幕上即顯示「印的參數」，詳如圖 5-1。

顯示參數現有值之後，**print**即詢問使用人是否重更改該值。如要改變各值，程式即向使用人索取各參數之值。如同第三章內的**gaslog** 程式一樣，前面的值是缺設答案；使用人只需鍵入需要更改的答案即可。

許多參數都是自我說明的。**page length** 就是每頁的行數；其初值為 66。**left margin**即各行開始印的欄數；初值是 15。（排印的頁行數自 0 起，左邊緣為 0 表自第一行開始印。）**right margin** 初值 132，並不能將其擴大。

很多的 PASCAL 編譯器允許一程式的原始本文，分開成若干不同的本文檔。在 APPLE 和 UCSD 的 PASCAL 的編譯指引

(\$I SPROCKET.TEXT)

促使編譯器包括 **SPROCKET.TEXT** 檔；其淨效益彷彿是該 **SPROCKET.TEXT** 已在程式本文之內。

```
*****
* print *
*****

Current print parameters:
Page length(lines):66
Left margin:15          Right margin:132
Print included files(Y/N):N
Output to file or printer (F/P):P
Output file name:
Printer init. string:@eP
Printer term. string:@eP
Any changes?(Y/N):_
```

圖 5-1 列印的最初顯示

print 可用來排印程式原始本文，它提供一個選擇，可以在本文原始檔中指明包含指引，來排印任意檔。爲了呼叫這個選擇，**print included files (Y/N)** 參數即設定 **<Y>**；否則設定爲 **<N>**。**print** 允許 **include** 指引呈巢狀結構。

因爲，並非所有的使用人都有列表機，**print** 可選擇重新指導其輸出一本文檔。爲了保持這個優點，使用人可在 **output to file or printer (F/P)** 顯示時，指明參數爲 **<F>** 值；**<P>** 表把輸出檔送到列表機。如該檔是本文檔，就向使用人索取檔案名稱。

許多列表機有很多特性，諸如縮小字體，粗體，斜體字，等等。這些特性可藉一序列字元當做命令來決定排印字體。例如，EPSON LX-80 列表機用 **<ESC><P>** (ASCII 27, ASCII 80) 做縮小字體的排印命令 (每吋 16.5 個字元)。**<ESC><Q>** 表正常字元 (每吋 10 字元)。

print 可讓使用人指明二個命令字元序列：開始時用初值化的命令輸送到列表機；第二是一個「終止」命令，在排印運轉結束時，將該命令回轉給列表機。

如何在這些字串中表示非排印字元呢？我們使用一個衆所周知的方法 **escape sequence**。首先，定義 **<@>** 做爲逸出 (**escape**) 字元。(不要和 **<ESC>** 搞混)。這個字元表示下列若干字元不可解釋成正常的本文。**print** 解釋逸出序列的規則如下：

- 在命令字串中的 **@e** (或 **@E**) 序列，將 **<ESC>** 送往列表機。
- **@C <char>** (或 **@C <char>**) 序列將一個控制字元送到列表機。**<char>** 表示一個字母字元。例如，**@cf** 被翻譯成 **<control-F>** (ASCII 6)；**@CS** 則爲 **<Control-s>** (ASCII 19)。

- **@hnn** (或**@Hnn**) 序列, (nn 值表 16 進位值。)

送一 ASCII 之值為 nn 到列表機。nn 可能是 0, 1, 或 2 個 16 進位數位。例如, 字串**@hiF** 送一個 ASCII 31 字元到列表機 (IF hex = 31 decimal)。

- **@@** 可能是把 @ 送到列表機。這是一特殊規則: 如果跟在 **<@>** 之後的字元不是 **<e>**, **<c>**, 或 **<h>** (或大寫等值) 等, 則解碼常式即將 **<@>** 刪除。 (於是, **@Zks** 變成 **Zks**)

- **@hoo** 表示一個空字串: 表示沒有字元要送到列表機去。

使用人根據自己的意願修訂上述參數之後, **print** 即要求鍵入付印的檔案名稱。檔案最多可以有 14 個。如果使用人只按 **<RETURN>**, **print** 就不再詢問檔案名稱了。當程式問 **Any changes ? (Y/N)**: ; 如果回答 **<Y>**, **print** 重新要求檔案名稱, 並將使用人以前的回答當做缺設。

鍵入檔案名稱之後, **print** 即顯示訊息:

please check printer 提醒使用人檢查列表機是否已準備好, 可以接受資料、開始排印。每個檔案被排印時就在螢幕上檔案名稱旁顯示 **← print**。印完之後會看到 **← done** 信號。如果打開檔、或是排印檔案時發生錯誤, 即顯示適當的錯誤代碼。圖 5-2 即螢幕上出現排印檔案的個數。

列出所有檔案之後, **print** 即問 **All done ? (Y/N)**:

。如果回答 **<N>**, **print** 即回到參數選擇部份, 允許在不離開程式之下排印檔案。

print 主常式如下:

```
{ $s+, v- }
program print;
{ Print textfiles }

uses
  { $u apple2:toolstuff.code } crtstuff, fixstuff, datestuff, textstuff;

const
  MAXPLEN = 999;      { Maximum lines on page }
```


• 5 本文檔案工具 •

```

MAXFILES = 14;      ( Maximum number of files user can specify per run )
PCOMSIZE = 15;      ( Max length of printer command strings )
MARGIN1 = 2;        ( # lines between top of page & header )
MARGIN2 = 1;        ( # lines between header & first text line )
MARGIN3 = 3;        ( # lines between last text line and bottom of page )
DATEWID = 8;        ( Width of date string )
LSTRSIZE = 255;     ( Long string size )

type
  pcommand = string[PCOMSIZE];

var
  name: array [1..MAXFILES] of filename;
  today: date;
  pagelen, rmarg, lmarg, nfiles, margin: integer;
  include, hardcopy: boolean;
  pinit, pterm: pcommand;
  outname: filename;

(-----)
( Modules to be inserted here: )
(      initprint                )
(      changeparams            )
(      getnames                )
(      printfiles              )
(-----)

begin ( print )
  crt(CLEAR);
  disptitle('print');
  initprint;
  repeat
    changeparams;
    getnames;
    printfiles
  until ask('All done?(Y/N):', MAXCRTROW - 1, FALSE, FALSE);
  crt(CLEAR)
end.

```

```

*****
* print *
*****

Enter up to 14 files to print.
Press <RETURN> when done

1:COLETTE.TEXT           - Done
2:KOALA.TEXT             - Done
3:ARCHIVE\FINDINCDIR.TEXT - Not Found
4:APPLE2\CRTSTUFF.TEXT   - Printing
5:LIB2\INVOICE.TEXT
6:LIB2\GAMESTATE.TEXT

```

圖 5-2 列印的動作

程式上方的編譯指引 { \$ St , V - } , 打開編譯的調換模 (SWAP mode) , 關閉編譯的字串長度型態查核。這個選擇詳見第三章。

常數 LSTRSIZE 指明 print 可以處理的最長字串, 我們選定為 APPLE 字串的上限 255。LSTRSIZE 限制輸出行的寬度, 我們亦冀望它儘可能的調整列表機, 可以排印很長的行。

全盤變數初值化 (Initializing Global Variables)

initprinter 程序自使用者那取得流程資料、並將 print 的其餘全盤變數初值化：

```
procedure initprint;
{ Initialize print's global variables }

var
  i: integer;

begin { initprint }
  i := (MAXCRTCOL - 20) div 2;
  posstr('Today's Date?:', i, 4);
  getdate(today, i + 14, 4, today, FALSE);
  pagelen := 66;
  lmarg := 15;
  rmarg := 132;
  include := FALSE;
  pinit := 'DeP';
  pterm := 'DeQ';
  outname := '';
  hardcopy := TRUE;
  margin := (MAXCRTCOL - 39) div 2;
end;
```

值的展示適用於 Epson MX - 80 上排印本文檔。列表機命令字串打開 MX - 80 縮小字體排印字元、來排印檔案, 並且結束時回轉列表機正常大小的字元。

一個比較複雜的 print, 是希望在磁碟上有一個小資料檔「記憶」參數設定值。開始的時候, print 從檔案資料取到前

一部份的參數。如果任何參數被改變，print 將新的值存到資料檔。在 APPLE PASCAL 的使用上，主要的困難是指定磁碟機號碼。

改變排印參數 (Changing Print Parameters)

利用 **Changeparams** 顯示流程參數，並接受使用人改變

它們：

```

procedure changeparams;
{ Allow user to change global printing parameters }

var
  s: string;

{-----}
{ Modules to be inserted here: }
{      getint      }
{-----}

begin { changeparams }
  center('Current print parameters:', 4);
  crt(ERASEOS);
  ftostr(pagelen, 0, 0, s);
  posstr(concat('Page length (lines):', s), margin, 6);
  ftostr(lmarg, 0, 0, s);
  posstr(concat('Left margin:', s), margin, 8);
  ftostr(rmarg, 0, 0, s);
  posstr(concat('Right margin:', s), margin + 23, 8);
  if include then
    s := 'Y'
  else
    s := 'N';
  posstr(concat('Print included files(Y/N):', s), margin, 10);
  if hardcopy then
    s := 'P'
  else
    s := 'F';
  posstr(concat('Output to file or printer (F/P):', s), margin, 12);
  posstr(concat('Output file name:', outname), margin, 14);
  posstr(concat('Printer init. string:', pinit), margin, 16);
  posstr(concat('Printer term. string:', pterm), margin, 18);
  while ask('Any changes(Y/N):', MAXCRTROW - 1, FALSE, TRUE) do begin
    pagelen := getint(margin + 20, 6, MARGIN1 + MARGIN2 + MARGIN3 + 3,
                      MAXPLEN, pagelen, TRUE);
    lmarg := getint(margin + 12, 8, 0, LSTRSIZE - (MAXFNAME + DATEWID + 1),
                   lmarg, TRUE);
    rmarg := getint(margin + 36, 8, lmarg + MAXFNAME + DATEWID + 1, LSTRSIZE,
                   rmarg, TRUE);
    include := getboolean(margin + 26, 10, include, TRUE);
    getstring(s, 1, margin + 32, 12, s, ['P', 'F'], TRUE);
    hardcopy := (s = 'P');
    if hardcopy then begin
      getstring(pinit, PCOMSIZE, margin + 21, 16, pinit, ['.', '\', ''], FALSE);
      getstring(pterm, PCOMSIZE, margin + 21, 18, pterm, ['.', '\', ''], FALSE);
    end
  end
end

```

```
        else ( file output )
            gettfname(outname, margin + 17, 14, outname)
        end
    end;
```

changeparam 在使用人指定輸出檔案時，會詢問輸出檔案名稱。如果使用人指定輸出檔輸送到列表機，它只要求列表機的初值和終止字串。

使用 **getint** 函數即能自使用者取得整數。

```
function getint(col, row, mini, maxi, defi: integer; defaulted: boolean): integer;
( Get integer from user )

var
    f: fixed;

begin ( getint )
    getfixed(f, col, row, mini, maxi, defi, 0, defaulted);
    getint := trunc(f)
end;
```

前面的工作是將定點數值輸入盡可能具有彈性，其結果是使得 **geting** 很容易寫。

取得檔案名稱 (Getting the File Names)

getnames 常式取得使用人希望要印的檔案名稱：

```
procedure getnames;
( Get file names to print )
"
var
    i: integer;
    s: string;

begin ( getnames )
    fto5(MAXFILES, 0, 0, s);
    center(concat('Enter up to ', s, ' files to print.'), 4);
    center('Press <RETURN> when done', 5);
    crt(ERASEOS);
    for i := 1 to MAXFILES do begin
        gotoxy(margin, i + 6);
        write(i: 2, ':');
        name[i] := ''
    end;
```

```

end;
repeat
  i := 0;
  repeat
    i := i + 1;
    gettfname(name[i], margin + 3, i + 6, name[i])
  until (name[i] = '') or (i = MAXFILES);
  if name[i] = '' then
    nfiles := i - 1
  else
    nfiles := MAXFILES
  until not ask('Any changes?(Y/N):', MAXCRTROW - 1, FALSE, TRUE);
  gotoxy(0, nfiles + 7);
  crt(ERASEOS)
end;

```

排印檔案 (Printing the Files)

print files 常式排印指定的檔案：

```

procedure printfiles;
{ Print files }

const
  PRINTER = 6;          { Unit number of printer }

var
  junk, status: iostatus;
  outf: tfile;
  outrec: tfreq;
  pageno, lineno, i: integer;

{-----}
{ Modules to be inserted here: }
{      decode      }
{      initoutput  }
{      termoutput  }
{      wrstr       }
{      wchars      }
{      skip        }
{      printfoot   }
{      fprint      }
{-----}

begin { printfiles }
  if initoutput(status) = 600010 then begin
    for i := 1 to nfiles do begin
      pageno := 1;
      lineno := 1;
      posstr('<- Printing ', margin + 27, i + 6);
      if fprint(name[i], pageno, lineno, status) = 600010 then
        posstr('<- Done      ', margin + 27, i + 6)
      else if (status = 9) or (status = 10) then

```

```

        posstr('<- Not found', margin + 27, i + 6)
    else if status = 7 then
        posstr('<- Bad name ', margin + 27, i + 6)
    else { something else bad happened }
        posstr('<- I/O Error', margin + 27, i + 6);
    if lineno > 1 then
        junk := printfoot(pageno, lineno, pagelen, junk)
    end;
    junk := termoutput(junk)
end
end;

```

本檔案包括有關本文檔案的輸出／入錯誤型態。如果想要接達一個磁碟（或一個「量」），而該磁碟（或 volume）不存在，APPLE PASCAL 的 `ioresult` 函數回轉數值 9。如果程式希望接達一個不存在的檔案，即回轉數值 10。若檔案名稱不合語法，即回轉數值 7。以上都是最普通的錯誤，如發生特殊狀況，會回轉特殊的錯誤訊息。其他錯誤都會與 `<- I/O Error` 訊息一同出現。（如果你覺得 I/O 錯誤訊息太不友善，或是資訊不夠，就把代碼更改之。可能的一種作法是，寫一個常式將一個 I/O 數字，回轉成一個描述錯誤的短字串，而該字串可以單獨傳送給使用者。）

輸出初值和終止 (Output Initialization And Termination)

`printfiles` 常式呼叫 `initoutput` 來定「輸出裝置」的初值，它將定了初值的字串送到列表機，或是打開特定的輸出檔。所有的檔排印好了之後，`printfiles` 呼叫 `termoutput`，它關閉輸出檔，或將終止字串送到列表機。

`initoutput` 常式：

```

function initoutput(var status: iostatus): iostatus;
{ Initialize printer or output text file }

{-----}
{ Modules to be included here: }
{      initprinter      }
{-----}

begin { initoutput }
    if hardcopy then begin
        if initprinter(pinit, status) <> G00010 then

```

```
        remark('Can't access printer')
    end
    else begin { file output }
        if tfcreate(outf, outrec, outname, status) <> GOODIO then
            remark(concat('Can't create ', outname))
        end;
        initoutput := status
    end;
```

這是一個簡單的二路決定，其決定則依布耳變數 **hardcopy** 的值而定。通常它回轉一個錯誤代碼、函數結果和參數 **status**。
initprinter 函數是：

```
function initprinter(pc: pcommand; var status: iostatus): iostatus;
{ Initialize printer }

begin { initprinter }
    remark('Please check printer...');
    {$i-}
    unitclear(PRINTER);
    status := ioresult;
    if status = GOODIO then begin
        decode(pc);
        if length(pc) > 0 then begin
            unitwrite(PRINTER, pc[1], length(pc), 0, 12);
            status := ioresult
        end
    end;
    initprinter := status
    {$i+}
end;
```

APPLE 和 UCSD PASCAL 有低位準 (low-level) 常式，直接控制周邊裝置。這些常式可被單元數字接達；系統排印的單元數字是 6，它由常數 **PRINTER** 定義之。呼叫

unitclear(unitnum)

將特定的單元重置成「電力接通」狀態 (power-up status)。在 **initprinter** 內使用它來測試在系統中是否有列表機，如果沒有，**ioresult** 回轉一個非零的值。

initprinter 和 **termprinter** 利用 **unitwrite** 內建程序

，把一序列的字串輸送到列表機。程序呼叫

`unitwrite(unitnum, buffer, length, blocknum, mode)`

自 `buffer` 寫 `length` 個位元組到單元數字 `unitwrite` 而 `buffer` 引數也許是轉移的起點位置。參數 `blocknum` 和 `mode` 是選擇的；如果它們沒有出現，可假定是 0。如該特定單元是一塊結構裝置，就像一個磁碟機，參數 `blocknum` 告訴 `unitwrite`，在該單元中那一塊的位元組可以被寫；如該單元是一串列裝置（serial device），像一列表機，則該參數無意義。

`mode` 參數更重要。本參數是一個控制 `<DLE>` 和 `end-of-line <CR>`（ASCII 13）字元的整數。如 `mode` 是 0，`<DLE>` 就被轉換成空白字元的個數。而 `<CR>` 被翻譯成 `<CR>—<LF>`。如 `mode` 是 12，則不作翻譯。於是，爲了避免意外的把字串轉換成列表機的命令字串，特定 `mode` 爲 12。

`initoutput` 工作時，`termoutput` 常式並不工作，它關閉輸出檔案或送終止字串到列表機。

```
function termoutput(var status: iostatus): iostatus;
{ Send termination command to printer or close output file }

{-----}
{ Modules to be included here: }
{      termprinter      }
{-----}

begin { termoutput }
  if hardcopy then begin
    if termprinter(pterm, status) <> GOODIO then
      remark('Can't close printer')
    end
  else begin { file output }
    if tfclose(outf, outrec, status) <> GOODIO then
      remark('Can't close output file')
    end;
    termoutput := status
  end;
```


termprinter 是：

```
function termprinter(pc: pcommand; var status: iostatus): iostatus;
{ Send termination string to printer }

begin { termprinter }
  status := G00010;
  decode(pc);
  if length(pc) > 0 then begin
    {$i-}
    unitwrite(PRINTER, pc[1], length(pc), 0, 12);
    status := ioreult;
    {$i+}
  end;
  termprinter := status;
end;
```

解碼逸出順序 (Decoding Escape Sequence)

initprinter 和 termprinter 二者皆呼叫 decode，它將列表機命令字串的逸出順序 (escape sequence) 翻譯成真正解碼字串，並將被翻譯的字串回轉到呼叫常式。

```
procedure decode(var pc: pcommand);
{ Decode printer command string }

var
  s: pcommand;
  i: integer;
  c: char;

{-----}
{ Modules to be inserted here: }
{      gesc      }
{-----}
begin { decode }
  s := '';
  i := 1;
  while gesc(pc, i, c) <> chr(0) do begin
    addchar(s, c, PCOMSIZE);
    i := i + 1;
  end;
  pc := s;
end;
```

decode 把個別的逸出順序解碼責任交付 **gesc**。而 **gesc** 注視命令字串的第 *i* 個字元；如該字元不是一個 **<@>** 逸出字元，它就回轉該字元，如果該字元是一個 **<@>**，就把後面的字元，當作是有意義的逸出順序。爲了要 **decode** 和 **gesc** 一同工作，**gesc** 必須留下 *i* 指向逸出順序的最後字元，以便下次透過環，**gesc** 會立刻注意後續順序。

```
function gesc(var s: string; var i: integer; var c: char): char;
{ Get (possibly escaped) character from string }

const
  ESCAPE = '@';      { Printer command string escape }

var
  c1: char;

{-----}
{ Modules to be included here: }
{      htoc      }
{-----}

begin { gesc }
  if gchar(s, i, c) = ESCAPE then
    if gchar(s, i + 1, c) = chr(0) then
      c := ESCAPE
    else begin
      i := i + 1;
      if c in ['E', 'e'] then
        c := chr(27)
      else if c in ['C', 'c'] then begin
        if gchar(s, i + 1, c1) in ['A'..'Z'] then begin
          c := chr(ord(c1) - 64);
          i := i + 1
        end
        else if c1 in ['a'..'z'] then begin
          c := chr(ord(c1) - 96);
          i := i + 1
        end
      end
      else if c in ['H', 'h'] then
        c := htoc(s, i)
      end;
  gesc := c
end;
```

gesc 使得大多數的逸出順序自行解譯 (interpretation)，它把十六進位數位解碼成字元，並交給 **htoc** 函數。像 **gesc** 一樣，**htoc** 留下 *i* 指向順序的最後字元。

```
function htoc(var s: string; var i: integer): char;
( Convert zero, one, or two hex digits into a character )

var
  n, nd: integer;
  c: char;

begin ( htoc )
  n := 0;
  nd := 0;
  while (gchar(s, i+1, c) in ['0'..'9', 'A'..'F', 'a'..'f']) and (nd < 2) do begin
    if c in ['0'..'9'] then
      n := 16 * n + ord(c) - 48
    else if c in ['A'..'F'] then
      n := 16 * n + ord(c) - 55
    else ( c in ['a'..'f'] )
      n := 16 * n + ord(c) - 87;
    nd := nd + 1;
    i := i + 1
  end;
  htoc := chr(n)
end;
```

htoc 最多超出 @h 順序二個字元。當它達到限度，或看一字元不能解譯成一個 16 進位數位，它即終止。

印各個檔案 (printing Individual Files)

一個檔案由 fprint 常式排印，非常的複雜。首先列出它的虛擬碼：

```
begin
  open input file
  while not end of file
    get line from file
    if at top of page
      print header
    print line
    if line didn't have a <CR> at end
      print a <CR> (fold long lines)
    if at bottom of page
      advance to top to next page
    if we're printing included files
      check line for include directives & print files if present
  end-while
  close input file
end
```

上述和 PASCAL 代碼之間的最大不同是錯誤查對。正在讀輸入或寫一輸出檔之際發生錯誤時，**fprint** 即出口，而不再做 I/O。

fprint 計算字串的最大長度，這個值適合我們定的邊際，**fprint** 把該值傳送給 **tfread**。當我們設計 **tfread**，如果沒有 <CR> 字元的任意字串長度大於特定的最大長度時，**fprint** 自動寫一個 <CR> 到輸出，以致於該字串其餘的部份在輸出檔後面若干行中出現。

如果布耳印字參數 **include** 是 **TRUE**，**fprint** 查對其輸入字串的包含指引。當每一指引被發現了，**fprint** 即遞迴呼叫本身，以排印特定的檔案。當被包含的檔案 (**included file**) 被排印時發生錯誤，就在螢幕的底端註記。這個方法比終止排印、或視同無意有用的多。

fprint 常式是：

```
function fprint(var name: filename; var pageno, lineno: integer;
                var status: iostatus): iostatus;
{ Print a file -- calls itself recursively for included files }

type
  lstring = string[LLSTRSIZE];

var
  intf: tfile;
  inrec: tfile;
  lstr, 's: lstring;
  i, max: integer;
  newname: filename;
  c: char;
  junk: iostatus;

{-----}
{ Modules to be inserted here: }
{      findincdir      }
{      printhead      }
{-----}

begin { fprint }
  max := rmarg - lmarg;
  ($r-)
  fillchar(lstr[1], lmarg, ' ');
  lstr[0] := chr(lmarg);
  ($r+)
  if tfopen(intf, inrec, name, status) = GOODIO then begin
    while status = GOODIO do
      if tfread(intf, inrec, s, max, status) = GOODIO then begin
```

```

s := concat(lmstr, s);
if lineno <= 1 then
  status:=printhead(today,pageno,rmarg,lmarg,name,lineno,status);
if status = GOODIO then
  if wrstr(s, status) = GOODIO then begin
    lineno := lineno + 1;
    if gchar(s, length(s), c) <> chr(13) then
      status := skip(1, status);
    end;
  if (status = GOODIO) and (lineno >= pagelen - MARGIN3) then
    status := printfoot(pageno, lineno, pagelen, status);
  if (status = GOODIO) and include then begin
    i := lmarg + 1;
    while findinmdir(s, i, newname) do
      if fprint(newname, pageno, lineno, junk) <> GOODIO then
        remark(concat('Error including ', newname))
      end
    end;
  if status = ENDFILE then { normal result }
    status := tfclose(intf, inrec, status)
  else { something bad happened }
    junk := tfclose(intf, inrec, junk)
  end;
  fprint := status
end;

```

代碼：

```

($r-)
fillchar(lmstr[1], lmarg, ' ');
lmstr[0] := chr(lmarg);
($r+)

```

是把 `lmstr` 的初值定為 `lmarg` 個空白字串，是一快速方法。

在 `fprint` 內可能有的錯（bug），包含在其自我呼叫之中（遞迴）。如果這個包含檔案的巢狀結構太深（直接包含、或其間包含），該程式最後會造成記憶不足、或被破壞。解決之道在於限制其遞迴深度，或在呼叫 `fprint` 之前查對記憶空間。

避免把一個長的列印字分成二部份，是一個簡單的改進。把這一長列結尾的部份存起來，並在下行列印時與結尾一同印出。（在該內容內定義一個「字」的良好方法是什麼？需要強迫限制字的最大長度嗎？）

其他的方法是，讓使用人按一鍵，來岔斷（interrupt）

排印。程式會向使用人是否希望繼續印流程檔、或略過去印下檔、或一起停印。

字串和字元輸出 (String and Character Output)

在 **initoutput** 和 **termoutput** 常式中，可利用布耳變數 **hardcopy** 直接把輸出資料送到本文檔，或是到列表機上。另外的例子 **wrstr**，把一字串送到本文檔、或列表機：

```
function wrstr(var s: string; var status: iostatus): iostatus;
{ Write string to printer or output file }

{-----}
{ Modules to be included here: }
{      lprint      }
{-----}

begin { wrstr }
  if hardcopy then
    wrstr := lprint(s, status)
  else { file output }
    wrstr := tfwrite(outf, outrec, s, status)
end;
```

如果字串將送往列表機，**wrstr** 即呼叫 **lprint** 工作之。

```
function lprint(var s: string; var status: iostatus): iostatus;
{ Write string to printer }

begin { lprint }
  {$i-}
  status := GOODIO;
  if length(s) > 0 then begin
    unitwrite(PRINTER, s[1], length(s));
    status := ioresult
  end;
  lprint := status
  {$i+}
end;
```

fprint 也呼叫 **skip**，它輸送特定數目的＜CR＞字元到輸出裝置：

```
function skip(n: integer; var status: iostatus): iostatus;  
( Skip n lines on output )  
  
begin ( skip )  
    skip := wchars(n, chr(13), status)  
end;
```

skip 呼叫 wchars 作輸出，wchars 爲一般性目的常式，
它反復單一字元特定次數到輸出裝置。

```
function wchars(n: integer; c: char; var status: iostatus): iostatus;  
( Write n characters to output )  
  
var  
    s: string[1];  
  
begin ( wchars )  
    s := ' ';  
    addchar(s, c, 1);  
    status := GOODIO;  
    while (n > 0) and (status = GOODIO) do begin  
        status := wrstr(s, status);  
        n := n - 1  
    end;  
    wchars := status  
end;
```

頁格式 (Page Formatting)

每輸出頁的開始，fprint 呼叫 printhead 排印各頁開始
的標題資訊：

```
function printhead(today: date; pageno, rmarg, lmarg: integer; name: filename;  
var linenos: integer; var status: iostatus): iostatus;  
( Print page header )  
  
var  
    dstr, pstr: string;  
    n: integer;  
  
begin ( printhead )  
    dtos(today, dstr);  
    ftos(pageno, 0, 0, pstr);  
    pstr := concat('Page ', pstr);  
    n := rmarg - length(dstr) - length(name) - lmarg;  
    if skip(MARGIN1, status) = GOODIO then  
        if wchars(lmarg, ' ', status) = GOODIO then
```

```
if wrstr(name, status) = GOODIO then
  if wchars(n, ' ', status) = GOODIO then
    if wrstr(dstr, status) = GOODIO then
      if skip(1, status) = GOODIO then
        if wchars(r marg - length(pstr), ' ', status) = GOODIO then
          if wrstr(pstr, status) = GOODIO then begin
            status := skip(MARGIN2 + 1, status);
            lineno := lineno + MARGIN1 + MARGIN2 + 2
          end;
        printhead := status
      end;
```

printhead 印出標點的各項目，由適當的空白和<CR>字分開。如果發生錯誤，它繼續查對，以保證不再做輸入／出。

每頁的結尾，**fprint** 呼叫 **printfoot** 跳到下頁的開端。（每個檔被列印之後，**printfoot** 也被 **printfiles** 呼叫，以致於下一個檔案在頁的頂端開始印。）大多數的列表機在下頁的頂端回答一個跳頁（<FF>，ASCII 12）字元。如果輸出的是一個檔案，**printfoot** 呼叫 **skip**，它輸出相當的空白列到檔案。

```
function printfoot(var pageno, lineno: integer; pagelen: integer;
                   var status: iostatus): iostatus;
{ Print bottom of page }
begin { printfoot }
  if hardcopy then { send formfeed to printer }
    printfoot := wchars(1, chr(12), status)
  else
    printfoot := skip(pagelen - lineno + 1, status);
  if status = GOODIO then begin
    pageno := pageno + 1;
    lineno := 1
  end
end;
```

找包含指引 (Finding Include Directives)

fprint 呼叫 **findincedir** 常式，以便尋找包含指引，呼叫

自字串 *s* 的第 *i* 字元起搜尋包含指引。如果找到了，函數回轉 **TRUE**，參數 *name* 包括被包含檔案的名稱，而 *i* 指向該指引之後的第一個字元。如果未找到，該函數回復 **FALSE**，而 *i* 指向字串尾端的下一字元。

findincdir 函數是：

```
function findincdir(var s: string; var i: integer; var name: filename):boolean;
{ Find include directive in string, return text file name }

var
  comment: string;
  c: char;
  j: integer;
  found: boolean;

{-----}
{ Modules to be included here: }
{      findcomment      }
{-----}

begin { findincdir }
  name := '';
  found := FALSE;
  repeat
    if findcomment(s, i, comment) then
      if gchar(comment, 1, c) = '$' then
        if gchar(comment, 2, c) in ['I', 'i'] then
          if not (gchar(comment, 3, c) in ['+', '-']) then begin
            j := 3;
            while gnbchar(comment, j, c) <> chr(0) do begin
              addchar(name, c, MAXFNAME);
              j := j + 1;
            end;
            found := (length(name) > 0);
            if found and (length(name) <= MAXFNAME - 5) then
              if (pos('.TEXT', name) = 0) and (pos('.text', name) = 0) then
                name := concat(name, '.TEXT');
          end;
        until found or (i > length(s));
      findincdir := found;
  end;
```

findincdir 常式利用 **find comment** 在字串中搜尋並且摘取 (**extract**) 註釋 (**comments**)。**findcomment** 回轉任何找到的註釋，剝掉註釋的定界標 (**delimiter**)。

註釋的第一個字元必須是 **< \$ >**，第二個字元是大寫或小

寫 $\langle I \rangle$, 第三個字元必須是 $\langle + \rangle$, 或 $\langle - \rangle$, 該包含指引語法才合法 ; 這種情況下 , 用 I/O 偵錯指引 ($\{ \$ i + \}$ 或 $\{ \$ i - \}$) 並不用包含指引。

如果所有的條件均被實行 , `findinmdir` 自字串搬移檔案名稱到變數 `name` 。像 `gettfname` , `findinmdir` 在字尾加上 `.TEXT` 。

`findcomment` 常式工作方法是 , 呼叫

```
found := findinmdir(s, i, name);
```

在字串的第 i 字元開始尋找註釋 , 如找到了 , `findcomment` 回轉 `TRUE` , 變數 `Comment` 包含註釋的本文 , 而 i 指向註釋的下一個字元。如果找不到 , `findcomment` 回轉 `FALSE` , i 指向字串尾端的下一個字元。

`findcomment` 是 :

```
function findcomment(var s:string; var i:integer; var comment:string):boolean;
{ Find Pascal comment in string }

var
  found: boolean;
  c1, c2: char;
  eoc, eos: boolean;
begin { findcomment }
  comment := '';
  found := FALSE;
  repeat
    if (gchar(s, i, c1) = '(') and (gchar(s, i + 1, c2) = '*') then begin
      i := i + 2;
      found := TRUE;
      repeat
        eoc := (gchar(s, i, c1) = '*') and (gchar(s, i + 1, c2) = ')');
        eos := (gchar(s, i, c1) = chr(0));
        if not (eoc or eos) then begin
          addchar(comment, c1, MAXSTR);
          i := i + 1;
        end
      until eoc or eos;
      if eoc then
        i := i + 2;
      else { end of string }
        i := i + 1;
    end
  else if gchar(s, i, c1) = '{' then begin
    i := i + 1;
    found := TRUE;
    while not (gchar(s, i, c1) in [']', chr(0)]) do begin
      addchar(comment, c1, MAXSTR);
    end
  end
end
```

```
        i := i + 1
      end;
      if c1 = ')' then
        i := i + 1
      end
    else
      i := i + 1
    until (i > length(s)) or found;
    findcomment := found
  end;
```

注意：**findcomment** 偵測註釋，包括中括號，或小括號和星號。**findcomment** 並不處理多行的註釋。有興趣的讀者可以想法子去證明。

建議 (Suggestions)

除了前面提到的建議之外，尚可考慮排印下列各種特性：

- 靠著列表機的速度和其介面，你可以注意列印行與行之間會暫停。這表示列印速度，較電子計算機產生要印的資訊為快。使用組合語言改寫「瓶頸常式、或改進其演算法，即能改進 **print** 的效率。
- 印各檔案時，顯示一些印程式的指標。這是很容易的，在排印時在螢幕顯示第一個第十行。或繼續顯示流程頁和列印行號。這對於 **print** 估計檔案大小有所助益，也可估計尚有多少行或多少頁待印。
- 允許使用者隨手插入報表紙和按 <RETURN> 的選擇，告訴 **print** 印下頁。排印每一行時，也允許使用人選擇暫停、插入一個不同的形式或紙張型態。

推薦閱讀 (Recommended Reading)

• 高等 Pascal 程式設計技巧 •

`print` 的設計取自 `Software Tools in PASCAL` (B.Kernighan 和 P.J.Plauger 著) 的中較為精緻的程式。你會發現若干使程式更具彈性的建議。也能看到若有價值的程式，它們使用本文檔作為輸入和輸出。

`6502 Assembly Language Programming` (Lance Leventhal) 和 `6502 Assembly Language Subroutine` (Lance Leventhal 著) 二書對於把 PASCAL 轉換成 6502 指令有所助益。

6

遊戲與戰略

(*Games and Strategy*)

遊戲對於電子計算機而言，非常風行，這些遊戲可能有下列型態：反應快、機智、簡單好運，適用於一個或多個對手，它的行為好像二個或多個玩家對抗的中庸仲裁者，它判斷他們的動作是否合法，並且擔任計分工作。

很多的電腦遊戲是屬心智技巧的競賽。

在部份領域之內，以機智取勝電子計算機，就像與 ferrari 作競足比賽一樣無望，任何電腦很輕易的取勝人們的傳統資料處理，或其他計算的任務，就以將 500 個名字用字母序分類，或 10 — 位數的乘法而言，人就無法與電子計算機比擬，類似這些例子，更是不勝枚舉。

因為電子計算機速度快，一些傳統的遊戲，諸如井棋（tic — tac — toe）或抄摸（nim）遊戲，都能處理的非常完美。電子計算機可以很快的分析出，各種移動棋子狀況的組合。

然而有許多出現狀況更複雜的遊戲，就算是最快的電子計算機，也無法檢定所有的情況，人類可以憑超越的智慧與知識玩這種遊戲，而這些技巧又不容易寫成程式，就像圍棋（go）、西洋棋。翻棋（Reversi）也非常難，本章將設計這個程

式。

翻棋的起源，並不很清楚；第一版大約 1980 年代，於英格蘭發展出來，據說是中國人發明的，然而法國人也很知道，最近才風行。

翻棋的規則 (The Rules of Reversi)

翻棋遊戲簡單易學，和西洋棋一樣，使用 8×8 的板子，棋子分為黑白二種。最初在板中各放二子，每一種顏色占一個斜線，見圖 6-1。

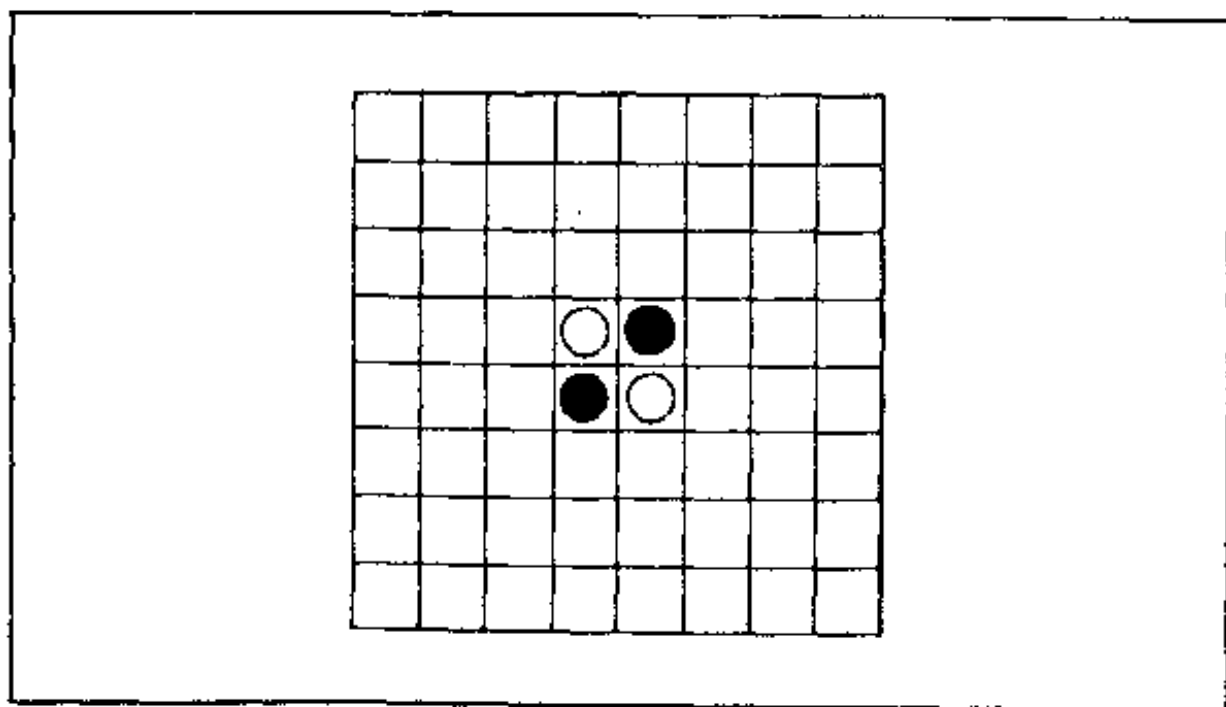


圖 6-1 棋的最初位置

玩家可以選擇黑色或白色，以投幣或引數作為遊戲的引數，傳統上黑色先著手。

玩家把自己的一顆棋子，下到空白的棋位上，把對方的棋子若干個夾住，於是把被夾住的顏色全部翻過來，變成自己棋

子的顏色，然後由對手下，方法相同，所有的變成見圖 6-2，而白色選擇著手後，見圖 6-3。

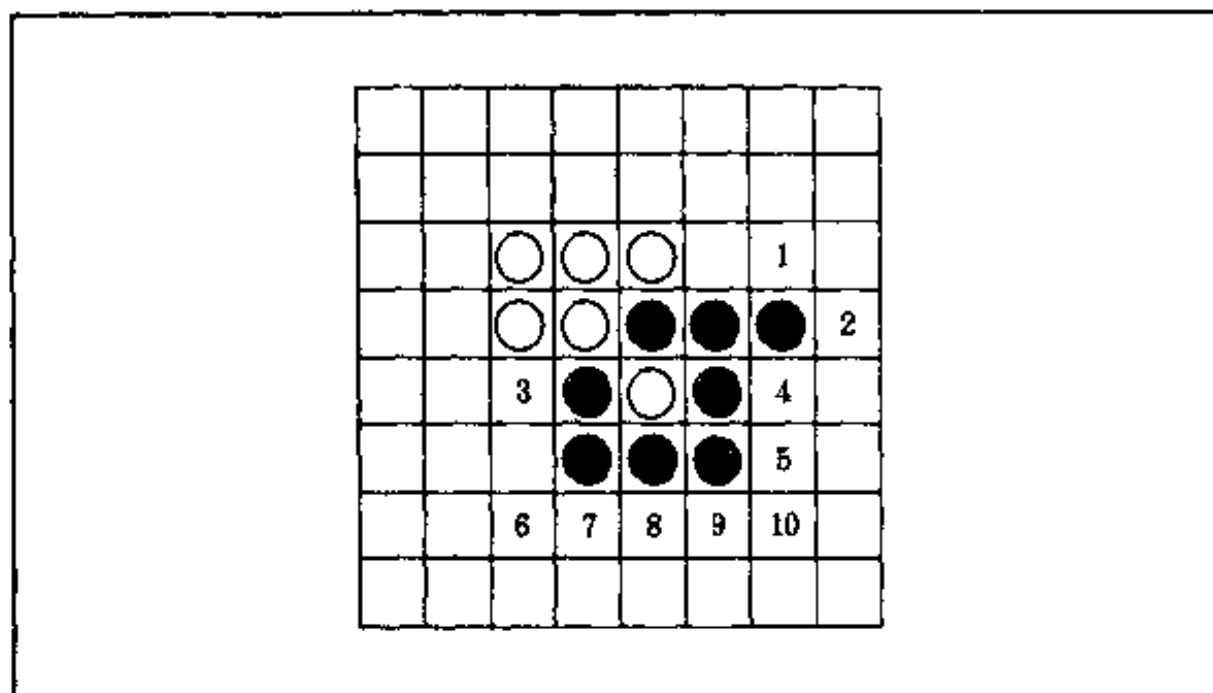


圖 6-2 白棋可以在註記號碼的格子上落子

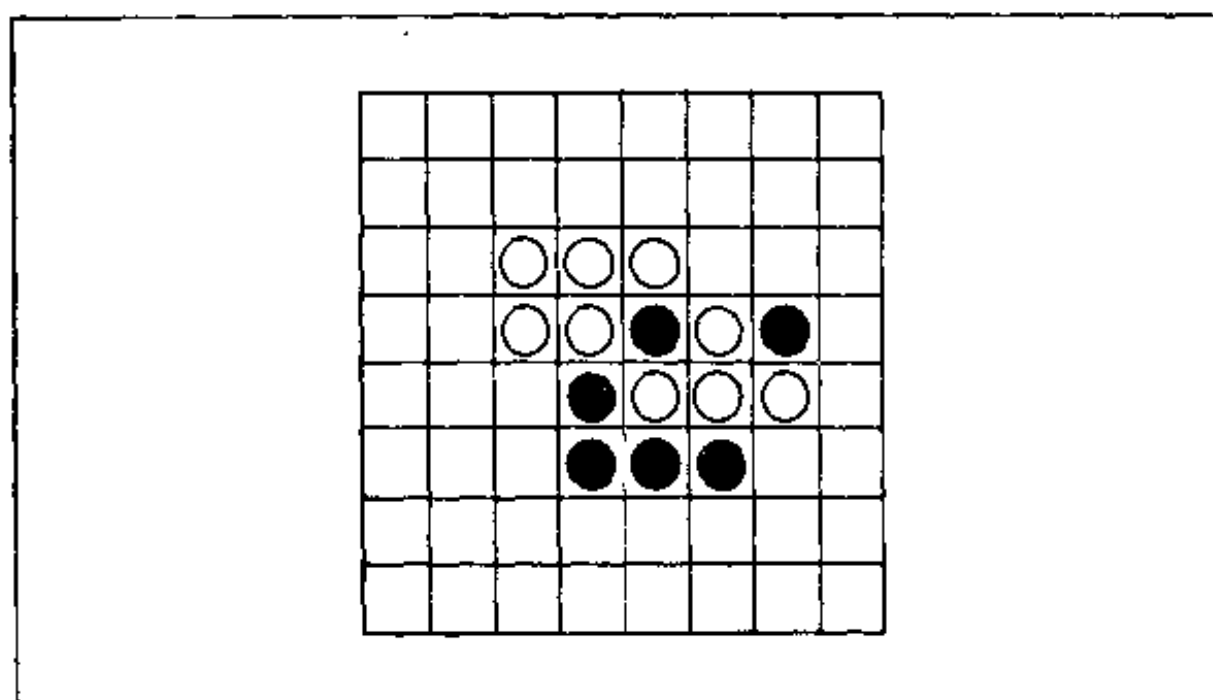


圖 6-3 白棋在上圖註記 4 的地方落子

兩人輪流著手，直到雙方無合規定的著手止，所謂無合規定著手，是指沒有空白處，或在空白處下棋沒有對方的棋子可翻，如果，只有一方有棋可下，另一方就可略過，放棄一手棋，下到結局，計算誰擁有較多的棋子，誰就獲勝。

運轉程式 (Running the Program)

我們的最基本目的，在於設計一個程式，適合一般的人作為對手，必須選擇一些好棋，避免惡手。此外 reversi 的速度要快的理由有二：第一、增加趣味性，玩遊戲的人通常沒有耐心去等。第二、在同一時段內，程式快得可以檢討更多的著手。

程式的使用方法也要簡單，人類對遊戲的本身遠較該程式的細節有興趣。好的遊戲程式，往往較真正的，使用棋盤的遊戲，更吸引人，因為電子計算機消除不合法的著手，不斷的計分，甚至於下出使用者不曾考慮過的著手。

reversi 與前面所提到的程式，在輸入方面的方法不同，所有的反應，都只用到一個鍵；程式以使用者按下的第一個鍵去檢查是否有效，並認為是使用人的回答。而 **<RETURN>** 並不當作輸入終止鍵，而 **<BACKSPACE>** 也不用作更正錯誤之用。

電子計算機首先顯示最初棋子在棋盤的位置，再詢問 **Do you want white or blank ? (W/B)**：回答 **<W>** 表選擇白棋，回答 **** 表示選擇黑棋，按其他鍵皆無效，並發出嗶嗶的聲音。

reversi 接著問：**Enter lookahead for computer (1-6)**：所謂 **lookahead** 表示考慮幾步棋的意思，第一次玩的人，多半選 1，棋力進步以後逐次增加。

回答上述二個問題之後，即開始比賽，每下一合法的著手，電子計算機即顯示其結果，全盤結束，它隨即宣佈誰獲勝。

輪到人下時，reversi 在空白方格顯示 \times 號，這星號的作用與游標相同，使用人就可按 $\leftarrow \rightarrow$ （其他機種請參考該機參考手冊），控制游標游走，每一次移動皆針對合法的著手，必要時尚可回到，第一次移動的地方，此可用 \leftarrow 循反向移動。這計劃是保證使用者的每一著手都合乎規定， \times 也只在合法的空白方格上出現。

選定適當的地方後，按 **RETURN** 鍵，電子計算即立刻更新畫面。

結局時，reversi 詢問 **play again ?(Y/N)**：使用人鍵入 **Y**，即重新玩下一局，鍵入 **N**，即離開本程式。

資料結構 (Data Structure)

第一個問題是，如何在電子計算機的記憶體內代表棋盤。

當程式掃描棋盤的不同方向的方格時，並非連續核對棋盤列陣是否超出範圍。

如果把這個 8×8 的棋盤，放大成 10×10 時，PASCAL 又如何表示呢？最清楚的解答是，以 2 維列陣來代表此一 10×10 的棋盤：

```
<type>
board = array [0..9, 0..9] of <something>;
```

如此安排對於程式的除錯和了解都有助益，是以早期的 reversi 版本，皆使用一個二維列陣，去代表記憶體內的棋盤。

儘管使用二維列陣的方法簡單，方法也很自然，對於二維

列陣內，接連一個個別的元素，遠較一維列陣內接連一個別元素困難。時間花費就大得多，在 APPLE 立上使用，就幾近多出一倍時間。

爲了很快找到相鄰，有效的空方格，使用一維列陣，也比較容易。所以，我們選擇一維列陣，來代表棋盤。

首先，定義常數和玩家的型態。

```
<type>
  contents = (LIGHT, DARK, EMPTY, BORDER);
  player = LIGHT..DARK;
```

用 LIGHT 和 DARK 來代表白棋和黑棋，（因爲 APPLE 繪圖常式，已將 WHITE 和 BLANK 納爲識別字，所以不再採用 WHITE 和 BLANK），EMPTY 表示空的方格子，棋盤方格的值是 BODEK。player 的型態，定義成 LIGHT 或 DARK。

棋盤方格編上 00 到 99 的號碼，以便利定義 squarenum 的資料型態。

```
<type>
  squarenum = 0..99;
```

對於 reversi 每一註上號碼的地方，都需要儲存一個表列，以供程式檢查，這個工作可由型態 movelist 完成之，茲定義如下：

```
<const>
  MAXMOVES = 60;

<type>
  movelist = record
    nmoves: 0..MAXMOVES;
    move: array [1..MAXMOVES] of squarenum
  end;
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

圖 6-4 棋盤編號表

movelist 掌握表列中移動棋子的數目 **nmoves** 和 **move** 陣列中的個別移動。**MAXMOVES** 定義表列的最大移動空間是 60，**MAXMOVES** 的值遠大於事實的需要。所以，我們移動表列的棋子時，即使增加數量，也不需去核對該表列是否超出範圍。

爲了控制棋盤，我們定義棋盤資料結構如下：

```

<type>
  board = record
    sq: array [sqarenum] of contents;
    ndiscs: array [player] of integer;
    possible: movelist
  end;

```

此處 **sq** 列陣，掌握棋盤方格的內容，列陣 **ndiscs** 則是每一個參與人員，擁有棋子的個數。**board** 記錄尚包含可能的合法的移動數目，記作 **possible**。如果某一方格是空的，

與其相鄰的又至少有一個以上的白子或黑子，我們就認為它是可移動的，每一個人的合理有效的移動，必然是可移動的部份集合。當找一個合理的著手時，就只需考慮該 **possible** 表列，每移動一次，**reversi** 就修正表列，以反映新的安排。

direction 型態，代表八種可能的水平的，垂直的，和斜線的方格，並以助憶符號表示：

```
<type>
direction = (NORTH, NORTHEAST, EAST, SOUTHEAST,
             SOUTH, SOUTHWEST, WEST, NORTHWEST);
```

翻棋的主常式 (Reversi — The Main Routine)

其次再設計 **reversi** 的本文版，它使得我們，在修正圖形之前，就能集中精神於程式的工作。設計本文版時，要牢記最終目標就是要使用這個圖案，這可以免除我們因為要擴大圖形，而重新設計程式的麻煩。

根據資料的定義，**reversi** 主常式設計如下：

```
($s+)
program reversi;
{ The game of Reversi }
uses
  ($u apple2:toolstuff.code ) crtstuff;

const
  MAXMOVES = 60;

type
  contents = (LIGHT, DARK, EMPTY, BORDER);
  player = LIGHT..DARK;
  squarenum = 0..99;
  movelist = record
    nmoves: 0..MAXMOVES;
    move: array [1..MAXMOVES] of squarenum;
  end;
  board = record
    sq: array [squarenum] of contents;
    ndiscs: array [player] of integer;
    possible: movelist;
  end;
  direction = (NORTH, NORTHEAST, EAST, SOUTHEAST,
               SOUTH, SOUTHWEST, WEST, NORTHWEST);
```

```

var
  accept, fwdkey, backkey, ch: char;
  delta: array [direction] of integer;
  sqord: array [squarenum] of integer;
  sqchar: array [contents] of char;
  corner, poison1, good1: array [1..4] of squarenum;
  poison2, good2: array [1..4, 1..2] of squarenum;
  edge: array [1..4, 1..4] of squarenum;
  xmarg, ymarg: integer;

{-----}
{ Modules to be included here: }
{      initrev                  }
{      dispgrid                 }
{      playagame                 }
{-----}

begin ( reversi )
  crt(CLEAR);
  disptitle('reversi');
  initrev;
  dispgrid;
  repeat
    playagame;
    center('Play again?(Y/N):', ymarg + 18);
    ch := getkey(ch, ['Y', 'N'], TRUE);
    eraseline(ymarg + 18)
  until ch = 'N';
  crt(CLEAR)
end.

```

設定全盤變數的初值 (Initializing Revesi's Global Variables)

在我們的大多數程式之中，取消一分離常式，以定翻棋的全盤變數之初值。由於APPLE PASCAL 編譯器限制任意單一的程式所產生目的的代碼的大小為 1200 位元組，常式被分開成兩個部份。**initrev** 程序如下：

```

procedure initrev;
{ Initialize reversi global variables }

procedure initrev1;
{ Initialize globals, part 1 }

var
  i, j, sv: integer;

begin ( initrev1 )
  accept := chr(13);
  backkey := chr(8);
  fwdkey := chr(21);

```

• 高等 Pascal 程式設計技巧 •

```

xmargin := (MAXCRTCOL - 32) div 2;
ymargin := 4;
sqchar[DARK] := 'B';
sqchar[LIGHT] := 'W';
sqchar[EMPTY] := ' ';
sqchar[BORDER] := '*';
sqord[11] := 1; sqord[12] := 7; sqord[13] := 2; sqord[14] := 2;
                    sqord[22] := 8; sqord[23] := 6; sqord[24] := 5;
                                sqord[33] := 3; sqord[34] := 4;
                                                sqord[44] := 0;

for j := 1 to 4 do
  for i := j to 4 do begin
    sv := sqord[10 * j + i];
    sqord[10 * i + j] := sv;
    sqord[10 * (9 - i) + j] := sv;
    sqord[10 * (9 - j) + i] := sv;
    sqord[10 * j + 9 - i] := sv;
    sqord[10 * i + 9 - j] := sv;
    sqord[10 * (9 - i) + 9 - j] := sv;
    sqord[10 * (9 - j) + 9 - i] := sv;
  end;
delta[NORTH] := -10;
delta[NORTHEAST] := -9;
delta[EAST] := 1;
delta[SOUTHEAST] := 11;
delta[SOUTH] := 10;
delta[SOUTHWEST] := 9;
delta[WEST] := -1;
delta[NORTHWEST] := -11;
end;

procedure initrev2;
( Initialize globals, part 2 )

var
  i: integer;

begin ( initrev2 )
  corner[1] := 11; poison2[1, 1] := 12; good2[1, 1] := 13;
  poison2[1, 2] := 21; poison1[1] := 22;
  good2[1, 2] := 31;
                                good1[1] := 33;

  corner[2] := 18; poison2[2, 1] := 17; good2[2, 1] := 16;
  poison2[2, 2] := 28; poison1[2] := 27;
  good2[2, 2] := 38;
                                good1[2] := 36;

  corner[3] := 81; poison2[3, 1] := 82; good2[3, 1] := 83;
  poison2[3, 2] := 71; poison1[3] := 72;
  good2[3, 2] := 61;
                                good1[3] := 63;

  corner[4] := 88; poison2[4, 1] := 87; good2[4, 1] := 86;
  poison2[4, 2] := 78; poison1[4] := 77;
  good2[4, 2] := 68;
                                good1[4] := 66;

  for i := 1 to 4 do begin
    edge[1, i] := 12 + i;
    edge[2, i] := 28 + 10 * i;
    edge[3, i] := 21 + 10 * i;
    edge[4, i] := 82 + i;
  end;
end;

begin ( initrev )
  initrev1;
  initrev2;
end;

```

initrev 定下列變數的初值：

- **accept** , **fwkey** , 和 **backey** 是使用者用來接受游標顯示搬移，把游標移動到下一個合法的表列，和將游標移動到前面一個合法移動表列。設定 **accept** , 所以按 **< RETURN >** 鍵即接受移動。**fwkey** 被設定成 APPLE 的 **< → >** 鍵 (**control-U**) , 而 **backey** 被設定成 **< ← >** 鍵 (**< BACKSPACE >** , **ASCII 8**) 。選擇這些鍵以反映使用者的特殊鍵盤引數。在 APPLE 上選擇的這些鍵，都是使用有意的助憶符號，此為相當有利的，在 APPLE II 的鍵盤上，它們靠的很近，這是不利之處：當使用者想按其他的一個鍵，可能會誤按 **< RETURN >** 。
- **xmarg** 和 **ymarg** 是在螢幕左上角顯示的 CRT 座標。選擇 **xmarg** 定螢幕水平的中央。**ymarg** 可將 CRT 減少顯示行，甚至於少於 24 行。
- **sqchar** 是一個陣列其所包含的字元，是用來顯示螢幕的每一棋盤的方塊：該方塊由黑棋占據時用字母 **< B >** 表示，白棋則由 **< w >** 表示；空的方格用空白表示。它的游標是星號字元 **< * >** 。
- **sqord** 是一個控制「由電子計算機所計算出來的次序」的列陣。當我們討論電子計算機如何移動時，我們將討論這個無用的特性。直到我們了解 **initrev** 只是指定 10 個方格的值，利用棋盤的八層對稱來指派值給其他的方格。
- **delta** 是一個列陣，包括把棋盤方格的索引 (**index**) 加相鄰方格的索引。例如，方格 44 其 **SOUTHEAST** 方向的方格是

$$44 + \text{delta}[\text{SOUTHEAST}] = 44 + 11 = 55$$

本法方簡單有效。

corner , **poison 1** , **poison 2** , **good 1** , **good 2** 和 **edge** 都是用來把不同的方格集合一起。例如，變數 **corner[2]** 是第二角的方格個數，假設為 18。**poison 1** 方格（ 22,27,72 和 77 ）是方塊垂直相鄰的角；**good 1** 方格（ 33,36,63 和 66 ）是二個與角不相鄰的方格。**poison 2** 方格（ 12,17,21,28,71,78,82, 和 87 ）是方塊的水平 and 垂直相鄰的角；**good 2** 方格（ 13,16,31,38, 61,68,83, 和 86 ）是二個與角的不相鄰的垂直和水平方向的方格。

所有的棋盤列陣是幫助電子計算機根據計算不同棋盤位置後的戰略。通常 **corner** , **good 1** 和 **good 2** 方格，用於當作好着，**poison 1** 和 **poison 2** 是考慮壞棋之用。這也是控制 **edge** 方格的好構想，後面將詳加討論翻棋的戰略。

顯示棋盤 (Display the Board)

程式一開始就該把棋盤反射到螢幕上，這個功能由

dispgrid 完成之：

```
procedure dispgrid;  
( Display board grid - text version )  
  
var  
  i: integer;  
  
begin ( dispgrid )  
  center('-----', ymargin);  
  for i := 1 to 15 do  
    if odd(i) then  
      center('!   !   !   !   !   !   !   !   !   !', ymargin + i)  
    else  
      center('!--!--!--!--!--!--!--!--!--!', ymargin + i);  
  center('-----', ymargin + 16)  
end;
```

簡易棋賽 (Playing a Single Game)

Playagame 程序與使用人對抗棋賽。決定棋賽結束與否的邏輯是有點技巧，其虛擬代碼作最好的描述如下：

```
begin
  initialize variables for game
  set current player to black
  set game-over flag to FALSE
  set previous-player-moved flag to TRUE
  repeat
    display current score
    if current player has a legal move
      get move from current player
      make move
      set previous-player-moved flag to TRUE
    else if previous player moved
      set previous-player-moved flag to FALSE
    else { neither player can move }
      set game over flag to TRUE
      set current player to other player
  until game over
end
```

「**previous-player-moved**」旗標指示使用人（玩家）的前一次着手。如果該着棋合規定，該棋標為**TRUE**，否則為**FALSE**。當玩棋的人無合規定的着手，而該旗標指示另一家在前一輪也無棋可下時，棋賽結束。

```
procedure playagame;
{ Play one game }

var
  mainboard: board;
  list: movelist;
  gameover, moved: boolean;
  computer, human, currentplayer: player;
  lookahead: integer;
  k: squarenum;

{-----}
{ Modules to be inserted here: }
{   dispsquare   }
{   setsquare   }
{   other       }
{   initgame    }
{   itos        }
{   disp_score  }
{   flanking    }
{   makelist    }
{   addmove     }
{   delmove     }
{   getmove     }
{   makemove    }
{   declarewinner }
{-----}
```

```
begin ( playgame )
  initgame;
  currentplayer := DARK;
  gameover := FALSE;
  moved := TRUE;
  repeat
    disp score;
    if makelist(list, currentplayer, mainboard) > 0 then begin
      moved := TRUE;
      k := getmove(list, currentplayer);
      makemove(k, currentplayer)
    end
    else if moved then
      moved := FALSE
    else ( Neither player able to move )
      gameover := TRUE;
      currentplayer := other(currentplayer)
  until gameover;
  declarewinner
end;
```

我們將參考主要棋盤，它用**mainboard** 變數代表。我們小心的自程式產生的部份，區分出主棋盤，以測試電子計算機的假設移動的結果。

定比賽變數的初值 (Initializing Game Variables)

每一局棋賽開始，**playgame** 呼叫 **initgame** 設定最初條件。

```
procedure initgame;
{ Initialize game variables }

var
  i, j: integer;
  ch: char;

begin ( initgame )
  eraseline(ymarg + 17);
  with mainboard do begin
    for i := 0 to 9 do begin
      sq[i] := BORDER;
      sq[i + 90] := BORDER;
      sq[10 * i] := BORDER;
      sq[10 * i + 9] := BORDER
    end;
    ndiscs[LIGHT] := 2;
    ndiscs[DARK] := 2;
    with possible do begin
      nmoves := 12;
```

geraseline 從圖形螢幕上抹除本文的水平線：

```
procedure geraseline(y: integer);
{ Erase line of text from graphics screen }

var
  i: integer;

begin { geraseline }
  pencolor(NONE);
  moveto(0, y);
  for i := 0 to XMAX div CHARWID do
    wchar(' ');
  end;
```

gposstr在圖形螢幕上的特定位置加一字串：

```
procedure gposstr(s: string; x, y: integer);
{ Put string on graphics screen }

begin { gposstr }
  pencolor(NONE);
  moveto(x, y);
  wstring(s)
end;
```

gcenter在圖形螢幕中央加一字串：

```
procedure gcenter(s: string; y: integer);
{ Center string on graphics screen }

begin { gcenter }
  geraseline(y);
  gposstr(s, (XMAX - length(s) * CHARWID - 1) div 2, y)
end;
```

最後，**gdisptitle** 在圖形螢幕上，把程式的標題放到盒子裡：

```
procedure gdisptitle(s: string);
{ Display title on graphics screen }

var
  ht, wd: integer;

begin { gdisptitle }
  ht := 10;
  wd := (length(s) + 1) * CHARWID;
  gcenter(s, YMAX - 10);
```

雜項的簡單常式 (Miscellaneous Simple Routines)

主棋盤在 **initgame** 中以簡單程序 **setsquare** 被設定初值。每設定一方格新值，也呼叫 **dispsquare** 來將新的引數在 **CRT** 螢幕上反射。

```
procedure setsquare(k: squarenum; c: contents);
{ Put piece on square }

begin { setsquare }
  mainboard.sq[k] := c;
  dispsquare(k, c)
end;
```

我們也呼叫 **setsquare**、透過 **reversi** 來改變方格的內容。

記住，我們還要修正程式，以便日後繪出圖形。在

dispsquare 內顯示一方格的內容：

```
procedure dispsquare(k: squarenum; c: contents);
{ Display square contents on text screen }

begin { dispsquare }
  gotoxy(xmarg + 4 * (k mod 10 - 1) + 2,
        ymarg + 2 * (k div 10 - 1) + 1);
  write(sqchar[c]);
  crt(HOME)
end;
```

而 **other** 函數在 **initgame** 被用來找顏色（使用人所選的顏色已知）。如果 **other** 的引數是 **LIGHT**，其回轉 **DARK**，反之亦然：

```
function other(pl: player): player;
{ Return other player's color }

begin { other }
  if pl = LIGHT then
    other := DARK
  else
    other := LIGHT
end;
```

other 指出利用純量型態來代表不同的值，有一利，亦有一弊。其利是，當設計師看到代碼

```
currentplayer := other(currentplayer)
```

顯然該程式是指現在輪到另一參與棋賽者的着手。

不利的是效率不好。每著一手棋，**other** 都要判斷「另一玩家」的顏色，我們只要不使用純量來定義方格的內容，而使用常數如下：

```
<const>
  LIGHT = 0;
  DARK = 1;
  EMPTY = 2;
  BORDER = 3;

<type>
  contents = LIGHT..BORDER;
  player = LIGHT..DARK;
```

其次避免使用 **other** 程序；例如，改用下列代碼

```
currentplayer := 1 - currentplayer;
```

但是這不如呼叫 **other** 來得清晰易懂。到目前為止，我們選擇比較清楚的純量型態。（雖然我們已注意到，更改它立即可以改進其效率。）

在每次移動棋子、顯示每個人棋子之前，**playagame** 呼叫 **dispscore**：

```
procedure dispscore;
{ display current score }

var
  s: string;

begin { dispscore }
  with mainboard do begin
    itos(ndiscs[computer], 2, s);
    posstr(s, xmarg + 10, ymarg + 18);
    itos(ndiscs[human], 2, s);
    posstr(s, xmarg + 31, ymarg + 18)
  end
end;
```

dispscore 利用 **itos** 將一整數轉換成一字串。**itos** 與第三章內描述的 **ftos** 相似。我們可以用 **ftos** 來取之，但是要把所有宣告部份定義成固定資料型態，詳細細節與 **reversi** 無關。

```
procedure itos(n, wid: integer; var s: string);
{ Convert integer to string }

var
  negnum: boolean;
  i, j: integer;
  ch: char;

begin { itos }
  negnum := (n < 0);
  n := abs(n);
  s := '';
  repeat

    addchar(s, chr(n mod 10 + 48), MAXSTR);
    n := n div 10;
  until n = 0;
  if negnum then
    addchar(s, '-', MAXSTR);
  while length(s) < wid do
    addchar(s, ' ', MAXSTR);
  i := 1;
  j := length(s);
  while i < j do begin
    ch := s[i];
    s[i] := s[j];
    s[j] := ch;
    i := i + 1;
    j := j - 1;
  end
end;
```

找合法移動 (Finding Legal Move)

在每次移動之前，**playagame** 呼叫 **makelist** 來為玩家組成一個合法移動的表列。這包括掃描主棋盤的 **possible** 移動表列，然後輸出現在玩家的合法移動。**makelist** 執行這個作用，並以一個 **var** 參數來回轉一個合法移動的表列，而且把

合法移動的數目作為其結果。

```
function makelist(var legal: movelist; pl: player; var bd: board): integer;
{ Make list of legal moves, return number of legal moves }

var
  i: integer;

{-----}
{ Modules to be inserted here: }
{      legalmove      }
{-----}

begin { makelist }
  legal.nmoves := 0;
  with bd.possible do
    for i := 1 to nmoves do
      if legalmove(move[i], bd, pl) then begin
        legal.nmoves := legal.nmoves + 1;
        legal.move[legal.nmoves] := move[i]
      end;
  makelist := legal.nmoves
end;
```

movelist 的結構和一字串相似：兩種結果皆有一任意數的元素（**movelist** 的移動，字串的字元），而其個數都有最大數的限制。也有指示器指示該結構內現有元素的個數（字串長度或 **movelist** 內的 **nmoves**）。為了簡化之，表列上有額外的限制：絕對不能有重複的著手。這種限制，也許不止檢查一次。由於在 **possible** 表列之中，沒有重複的移動，**makelist** 才能附加在每一合法的移動之後，並找遍合法的表列。

makelist 呼叫布耳函數 **legalmove**，測試特定的移動是否合法，如果合法回轉 **TRUE**，否則它就回轉 **FALSE**。

```
function legalmove(k: squarenum; var bd: board; pl: player): boolean;
{ Test if move is legal }

var
  ok: boolean;
  dir: direction;

begin { legalmove }
  dir := NORTH;
  ok := flanking(k, dir, bd, pl);
  while (dir <> NORTHWEST) and not ok do begin
```

```
        dir := succ(dir);
        ok := flanking(k, dir, bd, pl)
    end;
    legalmove := ok
end;
```

legalmove 利用 **flanking** 自被指明的方塊的方向繼續找，直到找不到為止，也就是說它的對手沿著某一固定的方向都被包圍了（該棋着手是合法的）或者是所有的方向皆檢查完為止（該棋的移動不合法）。呼叫

```
ok := flanking(k, dir, bd, pl);
```

檢查玩家 **pl** 是否包圍棋盤 **bd** 方向的方格 **k**。**flanking** 函數如下：

```
function flanking(k: squarenum; dir: direction; var bd: board; pl: player):
    boolean;
    (Return whether player flanks opponent from given square in given direction)
var
    ok: boolean;
    opponent: player;
    del: integer;
begin ( flanking )
    ok := FALSE;
    opponent := other(pl);
    del := delta[dir];
    k := k + del;
    with bd do
        if sq[k] = opponent then begin
            repeat
                k := k + del
            until sq[k] <> opponent;
            ok := (sq[k] = pl)
        end;
    flanking := ok
end;
```

flanking 檢查被指定的方向之相鄰方格。如果不是對手的棋子，則玩家並未包圍這個特別的方向。如果鄰居的格子是對手的，**flanking** 就繼續看同一方向的棋子，直到與其相交的方格，不是對手的為止；如果該方格，是玩家的棋子；玩家就在該方向包圍了對手的棋子。如果該方格是空的或是旁邊的

方格，則玩家在這個方向並未包圍對手的棋子。

最重要的是 **flanking** 相當的有效；它大約要檢查八次才能決定某一方格的着手是否合法。你也許考慮把 **flanking** 的資料結構改變。（如果用二維列陣代表棋盤會有何變化？如不採用棋旁邊哨，而採明顯的檢查方式，結果如何呢？）

取得玩家的着手 (Getting Players' Moves)

每當玩家的合法移動被編譯，**playagame** 即呼叫該 **getmove** 函數，便能得到玩家的某一移動（着手）。本函數是一個簡單的二向開關（**two way switch**），呼叫不同的常式以取得人的着手或電子計算機的着手：

```
function getmove(var list: movelist; pl: player): squarenum;
{ Get current player's move }

{-----}
{ Modules to be included here: }
{      gethuman      }
{      getcomputer   }
{-----}

begin { getmove }
  if pl = computer then
    getmove := getcomputer(list)
  else
    getmove := gethuman(list)
end;
```

二者之中 **gethuman** 比較簡單；使用人可藉 **<←>** 和 **<→>** 兩個鍵掃描合法着手，用 **<RETURN>** 來選擇：

```
function gethuman(var list: movelist): squarenum;
{ Get human's move }

var
  i: integer;
  ch: char;

begin { gethuman }
  i := 1;
  crt(BEEP);
```

```
with list do begin
  repeat
    dispsquare(move[i], BORDER);
    ch := getkey(ch, [accept, backkey, fwdkey], FALSE);
    dispsquare(move[i], EMPTY);
    if ch = backkey then begin
      i := i - 1;
      if i < 1 then
        i := nmoves
      end
    else if ch = fwdkey then begin
      i := i + 1;
      if i > nmoves then
        i := 1
      end
    end
  until ch = accept;
  gethuman := move[i]
end
end;
```

getcomputer 常式用來「計算」電子計算機的「最佳」着手。我們利用下個暫時的常式檢查程式的其餘部份。

```
function getcomputer(var list: movelist): squarenum;
{ Get computer's move - interim version }

begin
  getcomputer := list.move[1]
end;
```

本版的 **getcomputer** 只挑出着手表列的第一個着手，我們並不寄望這個簡易的演算可產生高明的玩法，但是很容易測試是否正確無誤。

下手棋 (Making the Move)

每當玩家選定一手棋，**makemove** 就被呼叫用來修正主棋盤，反射棋子的新引數：

```
procedure makemove(k: squarenum; pl: player);
{ Make move on main board }

var
  dir: direction;
  k1: squarenum;
  opponent: player;
  del: integer;

begin { makemove }
```

```

setsquare(k, pl);
opponent := other(pl);
with mainboard do begin
  ndiscs[pl] := ndiscs[pl] + 1;
  delmove(k, possible);
  for dir := NORTH to NORTHWEST do begin
    del := delta[dir];
    if flanking(k, dir, mainboard, pl) then begin
      k1 := k + del;
      repeat
        setsquare(k1, pl);
        ndiscs[pl] := ndiscs[pl] + 1;
        ndiscs[opponent] := ndiscs[opponent] - 1;
        k1 := k1 + del;
      until sq[k1] = pl;
    end;
    else if sq[k + del] = EMPTY then
      addmove(k + del, possible);
  end;
end;
end;

```

makemove 首先在被指定的地方，放置玩家的棋子，再從主棋盤的 **possible** 着手表列中把着手刪除之。再檢查八個所有的方向，把新包圍的對方棋子全部翻過來。如果新着手旁邊的方格是空的，**makelist** 就把它加列主棋盤的 **possible** 着手表列中。

addmove 常式即是把着手加到表列：

```

procedure addmove(k: squarenum; var list: movelist);
{ Add move to list, unless already present }
var
  i: integer;
begin { addmove }
  with list do begin
    move[nmoves + 1] := k;
    i := 1;
    while move[i] <> k do
      i := i + 1;
    if i = nmoves + 1 then
      nmoves := nmoves + 1;
  end;
end;

```

注意 **addmove** 搜尋已在表列中的元素，以確保其間沒有

• 高等 Pascal 程式設計技巧 •

重複。

delmove 與 **addmove** 相反；它自一表列中把着手刪除之：

```
procedure delmove(k: squarenum; var list: movelist);
{ Delete move from list }
var
  i: integer;
begin { delmove }
  with list do begin
    move[nmoves + 1] := k;
    i := 1;
    while move[i] <> k do
      i := i + 1;
    if i < nmoves + 1 then begin
      while i <= nmoves - 1 do begin
        move[i] := move[i + 1];
        i := i + 1;
      end;
      nmoves := nmoves - 1;
    end;
  end;
end
```

delmove 並不保證被刪除的着手，是真正的在表列之中。除非是它爲了要消除該着手才搜尋的；如果着手被找到了，並將之刪除，則 **nmoves** 減 1。

結束棋賽 (Finishing Up)

比賽完畢，**playagame** 呼叫 **declarewinner**，告訴使用誰是贏家。

```
procedure declarewinner;
{ Tell who won }
var
  diff: integer;
  s: string;
begin { declarewinner }
  with mainboard do
    diff := ndiscs[computer] - ndiscs[human];
  if diff > 0 then begin
    itos(diff, 0, s);
```

```
        center(concat('I won by ', s), ymargin + 17)
    end
    else if diff < 0 then begin
        itos(-diff, 0, s);
        center(concat('You won by ', s), ymargin + 17)
    end
    else
        center('We have tied!', ymargin + 17)
    end;
end;
```

雖然電子計算機的產生着手的演算法必須改進，它依然是 **reversi** 的完局常式。

比賽理論 (Game Theory)

reversi 如何才能決定合法着手中那一着棋才是妙着？在處理這特殊比賽戰略之前，先看看一般性的問題，並且描述一個方法，以期應用到本型態的任意比賽。

理論上，電子計算機可以檢查黑、白的所有可能的下法順序，並發現某種下法在完局可以獲勝。如果這種分析可以辦得到，則電子計算機總是能下出一盤「完美」的棋賽。對於簡單的遊戲而言，這種分析是可行的。

對於翻棋或更複雜的其他遊戲而言，這種理論是不切實際的，電子計算機要檢查的情況太多了。結果電子計算機考慮其未來的位置的次數受到限制。

爲了知道選擇某種着手可行，假定有二個人比賽，他們的名字分別是 **Maxie** 和 **Minnie**，而 **Maxie** 想要找到最好的一步棋。同時也假定，棋賽任何階段都能計分，數字愈大勝局機會愈大。

Maxie 的最簡單的戰略，是計算每一步棋的位置，最好的一步棋是棋盤中分數最高的位置。簡單的說，**Maxie** 企圖走一步棋，使其得分最高。

前面只考慮到未來的一着棋，進而考慮二着棋，亦即考慮

到Minnic 的可能下法，尤其是設下陷阱。

把Minnic 的反應一起計算，Maxie可能利用下列方法：在他下的每一步棋之後，即產生minnic 的下法，選擇其間最受歡迎的位置。本法在直覺上是很吸引人，但是並非最佳戰略，因為它假定Minnic 是按照Maxie 的意思下棋。假如Minnic 的智慧很高（或很幸運），也許下出Maxie 最不喜歡的一着棋。

我們可用圖形來解釋這個概念，詳如圖 6-5。上方的根節點（root node）代表最初的棋盤位置，下一層代表 Maxie 可能下的棋。底層表示Minnic 的可能下法。（簡單的例子，Maxie的合法下法有 4 種，而Minnic 也有 4 種。）底端的16 個節點代表棋盤位置被計算好的分數。

如果Maxie下出圖 6-5 中最左的分支（branch）。則 Minnie 選擇後，其分數可能是 2，9，7，或 9。因為高分代表Maxie的較佳位置，Minnic 選擇第一種着手，得分為 2。

若Maxie 選擇左邊第二個分支，Minnic 就從 8，5，3，5 等分數內選一個；選擇給予最低分的 3。同理，如果Maxie選擇左起第三分支，經Minnic 選擇後，給 Maxie 2 分，如果Maxie選左起第四分支，將得 1 分。

若此，Maxie的最佳選擇是什麼？顯然是第二個分支，因為Maxie 想得最高分，而Minnic 想使Maxie分數最低，Maxie 選擇這分支，可以保證最少 3 分。

本戰略可由minimax 了解，我們已討論了預料一、二步棋。如要預料三步棋，要預料Maxie 的最初選擇，然後是 Minnie，再預料Maxie 根據Minnic 之落子而着的第三步棋。很清楚地，這個工作只有電子計算機喜歡做。

預料的棋數愈多，要找出最好的移動方式，時間就愈長。

表 6-1 預料位置評估次數

Lookahead	Number of Positions Evaluated
1	8
2	64
3	512
4	4096
5	32768
6	262144

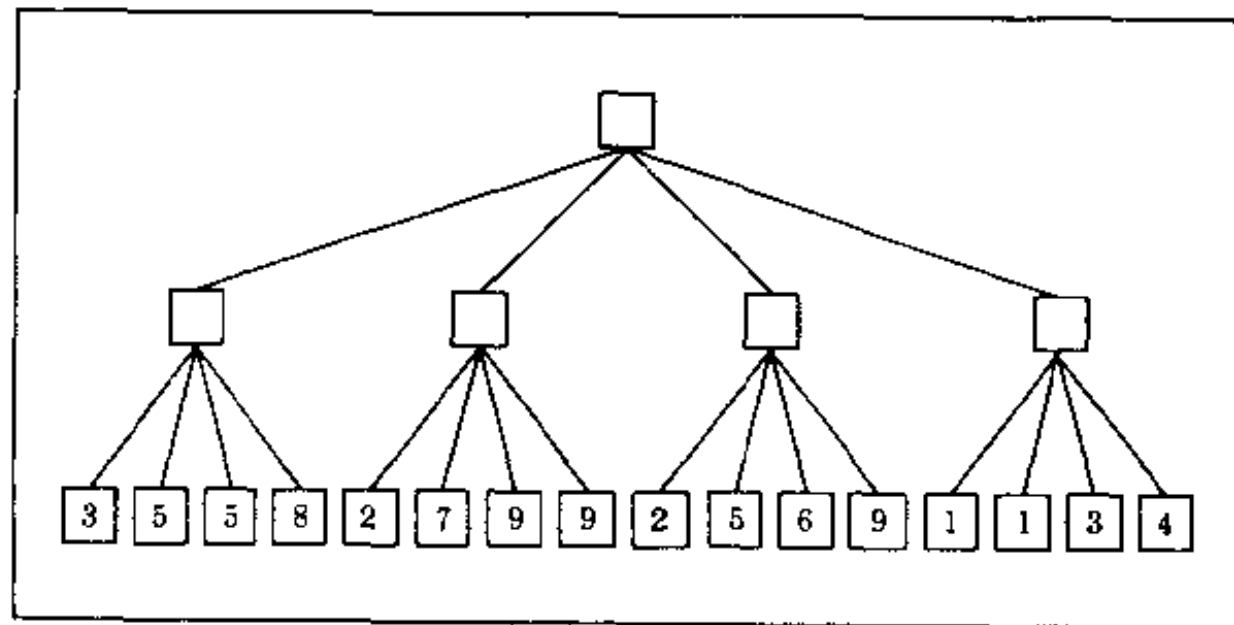


圖 6-6 節點排序樹狀結構

例如，每一輪迴每個玩家考慮八步棋，則考慮位置個數變得龐大。詳表 6-1。很幸運地，有一種簡單的方法，即 alpha-beta 修剪方法，可以減少 minimax 程序的運作時間。

爲了瞭解 alpha-beta 修剪工作，重新考慮圖 6-5 的樹狀結構。假定 Maxie 自左向右找最好的方法，當 Maxie 看到 Minnie 的前二種可能的棋路之後，他知道選擇第二個分支，至少可取得 3 分。當他看到 Minnie 的第二個回答，輪他走第

三個選擇，只得 2 分，Maxie 知道他不能選擇第三個分支；如果 Minnie 選擇第二種，他就能使 Maxie 走到一個較差的位置。Maxie 就不須考慮最後二種，這個分支就是經過修剪的。

同理，當 Maxie 發現 Minnie 第一次選擇之後，把第 4 種移動方法就只能得 1 分，他就不必再評估 Minnie 的其他選擇了。他知道他的第二個選擇，較第 4 種好。若是，他已避免看這 16 分支的五個。

也許有種簡易方式證明 $\alpha - \beta$ 修剪工作正確，即觀察 Maxie 的最佳着手是與被修剪的分支結果完全無關。無論我們把什麼值放到對應 Minnie 的第三或第四回答 Maxie 的第三種移動，或 Minnie 以第二、三、四回答 Maxie 的第四種移動，Maxie 的最好方法是第二分支。

通常，每當在任一個節點之下建立的捷進值，Maxie 都能停止計算 Minnie 的可能移動。（在前例中，捷進值為 3）。所謂捷進值係指相同水平的節之下，各最小值中取一最大值。

如果以上並未使你徹底迷糊，Maxie 尚可在 Minnie 之後，評估其本身的應手，評估時也採相似的修剪方式。當 Maxie 在前一個節的水平中看到諸最大值中最小的一個時，他即避免之，因為他知道 Minnie 會選擇前一個節而不選玩這個。有興趣的讀者可參考本章的結尾，繪出自己的樹狀圖，以便明白這處理工作。

儲存 $\alpha - \beta$ 修剪結果非常有戲劇性，然而其關鍵在於各被評估移動的次序，如圖 6-5 中由右向左評估，就無所謂修剪了。修剪方式愈好，則評估棋子移動速度愈快。例如，我們將原來的樹狀重新安排成圖 6-6。則 Maxie 可免除評估最後的九個位置：他只考慮 Minnie 針對他的第一種着手，她的 4 種回答。以及他選擇另三種時，她選擇第一種。

取得電子計算機的着手 (Getting the Computer's Move)

現在我們已找到電子計算機對人類下翻棋的最好着手，這種下棋法是根據minimax 演算而定的。爲了找出電子計算機的最佳着手，首先將寫兩種函數，findmax 和 findmin。

findmax 自所予的諸位置之中，找電子計算機的最好着手，則其可取得最大分數，同理，findmin 可根據最少的分數，而令人找到最好的一步棋。說得明白點，reversi 利用 findmin 互相呼叫。

新版本的getcomputer 像：

```
function getcomputer(var list: movelist): squarenum;
{ Get computer's move }

var
  max: integer;
  best: squarenum;

{-----}
{ Modules to be inserted here: }
{   eval                               }
{   trymove                            }
{   sortlist                           }
{   findmin fud decl.                  }
{   findmax                             }
{   findmin                             }
{-----}

begin { getcomputer }
  if list.nmoves = 1 then { only one legal move }
    getcomputer := list.move[1]
  else begin
    max := findmax(lookahead, list, mainboard, MAXINT, best);
    getcomputer := best
  end
end;
```

如果電子計算機僅走一步棋，在任何水平都無需再評估該遊戲的樹狀結構；getcomputer 只回轉其所下的那一步棋，否則該常式呼叫 findmax，爲電子計算機選擇最佳着手。

傳送到 **findmax** 的引數是預測該樹狀遊戲（電子計算機的合法着手數，現在的棋盤，捷進值），最好的着手由 **var** 參數 **best** 回轉之。當 **get computer** 呼叫 **findmax** 時，捷徑值是 **MAXINT**，這與未作修飾時一樣。只有在預測三或更多步棋 **findmax** 時才取捷徑值。

findmax 虛擬代碼是：

```
begin
  sort move list
  if there are legal moves
    repeat
      make next move in list
      if lookahead <= 1
        evaluate board
      else
        generate human's possible responses
        find score of human's best move
          (with one less move lookahead & cutoff of current maximum score)
        if score > previous maximum
          save new best move & maximum score
    until (last move has been considered) or (score is >= cutoff)
  else ( no legal moves )
    if lookahead <= 1 then
      evaluate board, set max score to result
    else
      generate human's possible responses
      set max score to score of human's best move
        (with one less move lookahead & no cutoff)
  return maximum score and best move
end
```

“ **Sort move list** ” 這個運算並未特別指明，但首先被考慮的是要找到電子計算機最佳着手，所以要先行按次序將着手排序。如前所說的，要改進搜尋的效率；將於後面詳述。

把虛擬碼翻譯成 **PASCAL** 如下：

```
function findmax(look: integer; var list: movelist; var bd: board;
  cutoff: integer; var bestmove: squarenum): integer;
( Get computer's best move )

var
  newlist: movelist;
  newbd: board;
  i, maxscore, score, nm: integer;
  junk: squarenum;

begin ( findmax )
  sortlist(list);
  with list do
    if nmoves > 0 then begin
```

```

maxscore := -MAXINT;
i := 1;
repeat
  newbd := bd;
  trymove(move[i], computer, newbd);
  if look <= 1 then
    score := eval(newbd, human)
  else begin
    nm := makelist(newlist, human, newbd);
    score := findmin(look - 1, newlist, newbd, maxscore, junk)
  end;
  if score > maxscore then begin
    maxscore := score;
    bestmove := move[i]
  end;
  i := i + 1
until (i > nmoves) or (maxscore >= cutoff)
end
else begin { no legal move }
  if look <= 1 then
    maxscore := eval(bd, human)
  else begin
    nm := makelist(newlist, human, bd);
    maxscore := findmin(look - 1, newlist, bd, -MAXINT, junk)
  end
end;
findmax := maxscore
end;

```

findmax 藉由變數 **newbd** 暫時複製一份現在的棋盤，據之估計電子計算的每一種變化，再藉呼叫程序 **stymove** 變出該着法。如果流程預見值是 1，**findmax** 即呼叫 **eval** 函數回轉一個分數給新的棋盤位置。否則它就在新棋盤中為玩家產生一個合法着手的表列，同時呼叫 **findmin** 回轉玩家最好的變法之分數。

因為 **findmin** 彼此呼叫，必須指明其中之一為 **forward** 程序；我們選定 **findmin** 如此定之，在 **findmax** 宣告成：

```

function findmin(look: integer; var list: movelist; var bd: board;
  cutoff: integer; var bestmove: squarenum): integer; forward;

```

findmin 看起來像 **findmax**，唯一不同的是 **findmin** 尋找最低分而非最高分，並且保留捷徑的邏輯。（當一個節比捷徑值低時就不再估計。）**findmin** 如下：

```

function findmin;
( Get human's best move )

var
  newlist: movelist;
  newbd: board;
  i, minscore, score, nm: integer;
  junk: squarenum;

begin ( findmin )
  sortlist(list);
  with list do
    if nmoves > 0 then begin
      minscore := MAXINT;
      i := 1;
      repeat
        newbd := bd;
        trymove(move[i], human, newbd);
        if look <= 1 then
          score := eval(newbd, computer)
        else begin
          nm := makelist(newlist, computer, newbd);
          score := findmax(look - 1, newlist, newbd, minscore, junk)
        end;
        if score < minscore then begin
          minscore := score;
          bestmove := move[i]
        end;
        i := i + 1
      until (i > nmoves) or (minscore <= cutoff)
    end
    else begin ( no legal moves )
      if look <= 1 then
        minscore := eval(bd, computer)
      else begin
        nm := makelist(newlist, computer, bd);
        minscore := findmax(look - 1, newlist, bd, MAXINT, junk)
      end
    end
  end;
  findmin := minscore
end;

```

findmin 和 **findmax** 都利用 **trymove** 來下棋 **trymove** 與 **makemove** 相似，除了該着手並不是在主棋盤下棋，而是由 **findmax** 和 **findmin** 建立一個暫時的棋盤，當然這些假設變法並不在螢幕上顯示：

```

procedure trymove(trysq: squarenum; pl: player; var bd: board);
( Try move, updating board )

var
  dir: direction;
  k1: squarenum;

```

```

    opp: player;
    del: integer;

begin ( trymove )
  opp := other(pl);
  with bd do begin
    sq[trysq] := pl;
    ndiscs[pl] := ndiscs[pl] + 1;
    delmove(trysq, possible);
    for dir := NORTH to NORTHWEST do begin
      del := delta[dir];
      if flanking(trysq, dir, bd, pl) then begin
        k1 := trysq + del;
        repeat
          sq[k1] := pl;
          ndiscs[pl] := ndiscs[pl] + 1;
          ndiscs[opp] := ndiscs[opp] - 1;
          k1 := k1 + del;
        until sq[k1] = pl;
      end;
      else if sq[trysq + del] = EMPTY then
        addmove(trysq + del, possible);
    end;
  end;
end;

```

估計一棋盤位置 (Evaluating a Board Position)

其餘的主要問題，是設計 **eval**，本函數接受一棋盤位置，回轉一個數值分數。唯一的要求是電子計算機希望 **eval** 回轉一較高的分數。

如前面討論的，最後誰擁有較多的棋子，誰就是贏家。這個建議把分數指派給新棋盤位置的方法是：只計算電子計算機的棋子個數。這種解法簡單、但非至善。往往玩家在棋賽過程中擁有多數棋子的優勢中，僅僅再變幾手棋却是勝負顛倒。因此，尚有其他因素左右玩家的位置是否好的。

其中一個因素是，玩家的方格之戰略值。雖然，我不能在這個主題中討論太詳細，我們很容易發現角的價值最大。只要玩家占據了某一角的方格，就不怕被翻成對手的顏色。

因為角部的方格價值高，玩家儘量避免走到與角相鄰的方

格。唯一的方法是讓另一方取得角的鄰居。走到前面 `initrev` 內討論的 `poison1` 或 `poison 2` 就喪盡優勢。

向後繼續變棋，希望讓玩家占據這些“`poison`”方格；因為這些方格，可能使對手走到這些位置的機會增加。

邊的方格也很重要。因為占據了一個邊，最多使對手占據另一邊。

除了考慮棋子個數和戰略值之外，每一個玩遊戲的人的活動性都得核算：遊戲的人能夠下的棋愈多，他或她的位置愈好。贏局的最普通的方法是強迫對手只有一種着法。

在設計評估的函數中，以上三種因素都列入考慮。愈是受電子計算機歡迎的着手，`eval` 回轉的分數也就愈高。

棋子的優勢最容易計算：人所擁有的棋子個數減掉電子計算機擁有棋子的個數即可。

活動性的因素也很容易變成數值的項目：計算其有效、合法的着手即可。該數處理因難與否和輪到電子計算機呢或是輪到人有關連。在電子計算機這一方的好棋、就是人這一方的壞棋。

戰略因素最難計量；用最簡單的方法來指派前述好棋與壞棋的分數。例如，電子計算機每得一角、增加 n 點，則人就減 n 點。如果一角是空的，該函數即檢視與角相鄰的方格是誰的，如屬電子計算機的，就減分數，人就加分。如果任意壞方格皆空，即檢視相對應的「好的」方格，為機器所有即加分，為人所有即減分。

最後，考慮四個邊：如果某一邊都是電子計算機的棋子，就列入加的因素，如果都是使用人的棋子，就考慮成減的因素。

把上述四個因素組合成一個單一數值分數是有幾分任性。我們選擇電子計算機占優勢的棋子個數與活動性的因素，乘以一個由程式指明為常數，並把它們加在一起。

• 6 遊戲和戰略 •

這一切可看 eval 函數：

```
function eval(var bd: board; pl: player): integer;
{ Evaluate board position }

const
  K1 = 1;      { weighting factor for disc advantage }
  K2 = 3;      { weighting factor for mobility }
  K3 = 200;    { score for owning corner }
  K4 = -100;   { penalty for owning poison1 square }
  K5 = 50;     { score for owning good1 square }
  K6 = -25;    { penalty for owning poison2 square }
  K7 = 15;     { score for owning good2 square }
  K8 = 10;     { score for having only discs on edge }
  K9 = 20;     { score for occupying edge }

var
  list: movelist;
  i, j, score: integer;
  c: contents;
  sideset: set of contents;

begin { eval }
  with bd do
    if ndiscs[human] = 0 then { Human wiped out }
      eval := MAXINT
    else if ndiscs[computer] = 0 then { Computer wiped out }
      eval := - MAXINT
    else begin
      score := K1 * (ndiscs[computer] - ndiscs[human]);
      if pl = computer then
        score := score + K2 * makelist(list, pl, bd)
      else
        score := score - K2 * makelist(list, pl, bd);
      for i := 1 to 4 do begin
        c := sq[corner[i]];
        if c = computer then
          score := score + K3
        else if c = human then
          score := score - K3
        else begin { corner empty, check poison squares }
          c := sq[poison1[i]];
          if c = computer then
            score := score + K4
          else if c = human then
            score := score - K4
          else begin
            c := sq[good1[i]];
            if c = computer then
              score := score + K5
            else if c = human then
              score := score - K5
          end;
        end;
        for j := 1 to 2 do begin
          c := sq[poison2[i, j]];
          if c = computer then
            score := score + K6
          else if c = human then
            score := score - K6
        end;
        c := sq[good2[i, j]];
        if c = computer then
          score := score + K7
      end;
    end;
```

```
        else if c = human then
            score := score - K7
        end
    end
end
end;
for i := 1 to 4 do begin
    sideset := [];
    for j := 1 to 4 do
        sideset := sideset + [sq[edge[i, j]]];
        if sideset = [computer] then
            score := score + K9
        else if sideset = [computer, EMPTY] then
            score := score + K8
        else if sideset = [human, EMPTY] then
            score := score - K8
        else if sideset = [human] then
            score := score - K9
        end;
    end;
    eval := score
end
end;
```

將變棋表列排序 (Sorting the Move Lists)

使用 alpha-beta 修剪的效率、與變法的次序相關；最好的狀況是，首先評估每一玩家的最佳着手。問題是事先無法預知何者最好；如果我們可以分辨出最佳着手，就不必搜尋比賽的樹狀結構。

不幸的很，隨意猜最好次序並無所助益。前面討論的，我們判斷戰略值，先評估角、壞的空格最後評估，其他方格在中間評估。sortlist 將着手排序，由 sqord 列陣放回 initrev 初值程序。

sortlist 利用殼式排序 (Shell sort) 將着手在表列重新排序。殼式排序是一種適用少量排序的好方法之一。

sortlist 程序如下：

```
procedure sortlist(var list: movelist);
{ Sort move list to put good moves first }

var
    i, j, jg, gap, k: integer;
begin { sortlist }
    with list do begin
        gap := nmoves div 2;
        while gap > 0 do begin
```


• 6 遊戲和戰略 •

```
for i := gap + 1 to nmoves do begin
  j := i - gap;
  while j > 0 do begin
    jg := j + gap;
    if sqord[move[j]] <= sqord[move[jg]] then
      j := 0
    else begin
      k := move[j];
      move[j] := move[jg];
      move[jg] := k
    end;
    j := j - gap
  end
end;
gap := gap div 2
end
end;
end;
```

有些 PASCAL 可能有一內建常式、或接達一般目的排序常式，比 `sortlist` 快。

衡量 (Measurements)

前面敘述 alpha-beta 修剪和着手表列排序，可以改進電子計算機的變棋搜尋常式。但是改進程度如何？`gethuman` 常式被改變而去呼叫 `findmin`，來找最好的回答，答覆由 `get-computer` 所產生的變法。

`reversi` 的本版速度有些改變(1)直接用 `minimax` 不用 alpha-beta 修剪來找最好的下棋方法。(2)使用修剪而不排序。我們度量這兩版的程式和原始版本使用時間，預見值為 1，2，和 3 來完成遊戲。

預見值為 1，alpha-beta 修剪和排序，都使遊戲時間略為增加。其原因是 alpha-beta 修剪對於預測一手棋的純 `minimax` 演算法無效；額外的測試只有增加困難。然而運轉時間增加是無意義的：利用 alpha-beta 修剪程式，運轉約較 `minimax` 版本慢 0.2%，而依序着手表列的方式較純 `minimax` 版本慢 2%。

只有預測二、三步棋才有實值的利益。預測二步棋時，alpha-beta 修剪比純 `minimax` 快了 42%。而排序比純的

minimax 節省了 65% 運轉時間。預測三手棋，alpha-beta 修剪比純minimax 節省 67%，排序與修剪連合使用則快了 79%。

因為 alpha-beta 減少估計位置的個數，此處所列的百分比與 eval 函數和其他程式比較有相當的關係。例如，如果 eval 速度加快，相對的百分比就減少。程式的絕對效率將改進。

這是個好消息。壞消息是縱然我們改進，預測較多的着手 reversi 是相當的遲鈍。預測三手棋，在APPLE II 平均約花二分鐘變出一手棋。這對於交作式的比賽而言簡直太慢了。

有兩種有益的研究：第一是改進移動尋找演算法，以考慮在樹狀結構中較少的移動。可能的方法很多，見本章最後。

第二種方法是老的常式改用組合語言。reversi 有若干個模組可用此處置。我們將限制翻譯一個常式 flanking。如前面所提的，該常式經常被呼叫，尤其是每當變一手棋（在 makemove 中），嘗試一手棋或是決定是否合法時。

flanking 用 6502 組合語言是：

```
.func flanking,4
;
; Routine to test whether player outflanks other in given direction
; from given square.
;
; declared in Pascal host as:
;
; function flanking(k: squarenum; dir: direction; var bd: board;
;                                     pl: player): boolean; external;
;
return .equ 0          ; return address
k      .equ 2          ; square number
dir    .equ 4          ; direction
bd     .equ 6          ; board address
pl     .equ 8          ; player color (0 or 1)
result .equ 0A         ; function result
other  .equ 0C         ; other player's color
del    .equ 0D         ; increment for specified direction

pop return      ; store return address
pla            ; discard stack bias
pla
pla
pla
```

```

pop pl          ; store player's color
pop bd          ; store board address
pop dir         ; store direction
pop k           ; store square number
;
; store FALSE result
;
lda #0
sta result
sta result+1
;
; calculate other player's color = 1 - pl
;
sec
lda #1
sbc pl
sta other
;
; get increment for specified direction
;
ldx dir         ; put direction in x-reg
lda delta,x     ; get increment
sta del         ; and store it
;
; calculate offset of neighboring square, store in y
;
lda k           ; get initial square number
asl             ; multiply by two
clc             ; prepare to add
adc del         ; add increment to get offset
tay             ; move it to y
;
lda (bd),y      ; get contents of neighboring board square
cmp other       ; is it other player?
bne exit        ; no, exit
nextsq          ; get old offset
tya             ; get old offset
clc             ; prepare to add
adc del         ; add increment
tay             ; move new offset to y
lda (bd),y      ; get contents of board square
cmp other       ; does square contain other player's piece?
beq nextsq      ; yes, go back to look at next square
cmp pl         ; does square contain player's piece?
bne exit        ; no, exit (border or empty square)
lda #1         ; store TRUE
sta result      ; in low byte of result
exit           ; push result
push result     ; push result
push return     ; push return address
rts            ; and return
;
; Table of increments (in hexadecimal)
;
delta          .byte -14,-12,2,16,14,12,-2,-16

```

注意這代碼利用 **push** 和 **pop** 互指令（在第五章內定義）。本組合語言的 **flanking** 是利用 PASCAL 的下列特性：

• 高等 P a s c a l 程式設計技巧 •

- APPLE P A S C A L 在 **board** 記錄的開始處儲存 **sq** 陣列，位址 **bd** 指著 **bd.sq [0]** 的內容；**sq** 陣列的每一元素花 2 個位元組。
- **del ta** 陣列對應 P A S C A L 代碼中的 **delta** 陣列。在組合常內所對應的元素都乘以 2（因為 **sq** 陣列的元素都占 2 位元組）。
- APPLE P A S C A L 的純量型態（**scalar type**）是 2 位元組的整數。第一個元素是 0，第二個元素是 1，等等。於是，組合語言常式在計算變數 **other** 中使用的識別符號 **LIGHT** 和 **DARK** 是 0 和 1。在檢定特定方向的增值時，該常式假設 **NORTH** 是 0，**NORTHEAST** 是 1，等等。
- 布耳值 **TRUE** 由整數 1 表示（或任意非零整數）；布耳 **FALSE** 值以 0 表示。一個布耳變數與一整數一樣占用二個位元組。

當使用組合語言的 **flanking** 來取代 P A S C A L 版本，我們衡量其速度上的改進；大約節省 60 %，於是預測三步棋，原來要 2 分鐘，如今只花 45 秒鐘，讀者可自行決定是否改寫其他常式。

APPLE 圖形初階（APPLE Graphics Primer）

在 P A S C A L 程式中使用的圖形命令，比其他非易傳性的特性更不具易傳性。由電子計算機硬體提供圖形功能很廣，然而不幸的是我們無法保證，在 APPLE II 上使用的，却不一定適用其他電子計算機——即便該機器有相似的圖形能力亦然。希望修飾機器的硬體的常式，以便自此處討論的得到一些指標。

APPLE PASCAL 提供一些常式，使得 PASCAL 程式充份利用 APPLE II 的繪圖功能。這些常式並非內建常式。例如，**length** 函數是內建函數；在編譯單元內為 **turtlegraphics** 呼叫，而 **turtlegraphics** 在系統館內。

APPLE 的圖形螢幕為一矩形，它解析度是（水平方向）280 點乘（垂直方向）192 點。原點（ $x = 0$ ， $y = 0$ ）在螢幕的左上角。（與本文螢幕的原點不同。）螢幕有六種顏色顯示功能：白、黑、橘、藍、紫、和綠。（此處僅使用黑白二色。）

APPLE PASCAL 圖形程序將分 4 部分：

開始／終止程序（**Initialization / Termination Procedure**）

呼叫程序 **initturtle**，將正常的本文螢幕轉成圖形螢幕，本命令同時還把圖形螢幕消除成黑色。程式呼叫 **textmode** 即能回到本文螢幕。

Line 圖程序（Line-Drawing Procedures）

線是利用圖形螢幕四周移動 **turtle**，而該 **turtle** 的位置由座標 X 和 Y 來指明。**turtle** 也可想像成一固定方向的面，其方向是由逆時針方向定出角度；角度為 0 表示面向右方、平行 X 軸。

移動 **turtle** 即能在螢幕上畫線，**movecd** 命令將在面向 **turtle** 的方向移動 d 單元。**move (x, y)** 是將 **turtle** 現在位置移到 (x, y) 。

turtle 可利用程序 **turn** 和 **turnto** 將 **turtle** 面向位置改變。**turn** 程序是一個相對轉向；**turn (t)** 即逆時針方向 t 度的方向。**turnto** 是絕對方向；**turnto (t)** 即轉 t 度，其方向依原來方向。

turtle 的繪圖顏色由 **pencolor** 程序控制。而 **pencolor**

(**WHITE**) 命令即在螢幕上畫白線，呼叫 **pencolor (NONE)** 即 **turtle** 的移動為不可見，雖然 **turtle** 可用任意的顏色，我們却只能有二種選擇。

再舉一例，下面的程序（在 **reversi** 中未使用）畫一矩形框：

```
procedure frame;  
{ Draw frame around boundary of screen }  
  
begin { frame }  
  pencolor(NONE);  
  moveto(0, 0);  
  turnto(0);  
  pencolor(WHITE);  
  move(279);  
  turn(90);  
  move(191);  
  turn(90);  
  move(279);  
  turn(90);  
  move(191);  
end;
```

繪圖形程序 (Picture-Drawing Procedure)

除了畫線之外，圖形常式尚提供一畫矩形圖的方法，該常式由 **drawback** 呼叫；如

```
drawblock(pic, rowsize, xskip, yskip, width, height, x, y, mode);
```

圖形由 **pic** 來註明其位置是在螢幕的 (**x** , **y**) 位置。**pic** 是個 2 度空間布耳值的包裝陣列；陣列的每一元素由圖形中的各點組成。**TRUE** 值則該點在圖中（顯示成白色），而 **FALSE** 值則不在圖中（顯示黑色）。

參數 **xskip** 和 **yskip** 自陣列的某一位置，抄到螢幕上。**width** 和 **height** 說明陣列抄到螢幕上點的個數。利用 **drawblock** 允許自陣開始把整個陣列抄到螢幕上。

rowsize 參數告訴 **drawblock** 在每一列中有多少個位元

組。該值由下列公式計算出：

```
rowsize      := 2 * ((rowdots + 15) div 16);
```

最後，**mode** 參數是介於 0 到 15 的整數。通常其為 10，它只將 **pic** 陣列抄到螢幕。

本文顯示程序 (**Teyt - Display Procedure**)

APPLE PASCAL 也提供程序在圖形螢幕顯示本文。呼叫 **wchar (ch)** 把 **ch** 字元放在螢幕 **turtle** 的流程位置上。呼叫 **wstring** 為字串 **s** 作相同的工作。

在 **turtlegraphics** 單元中提供一些常式，我們却只用到其中之一。

修飾reversi圖形 (**Modifying Reversi for Graphics**)

除非圖形開關改變重印程式的全部，我們只描述改變的本身。因為我們設計的很合理，只修飾初值設定的常式和執行輸出的部份，程式的其他部份不變。我們已設計一些對應本文螢幕工具的圖形工具，大部份的改變不過是 1 對 1 的替代。改變的有：

- 通知編譯器利用 **uses** 敘述自

```
uses
  ($u apple2:toolstuff.code ) crtstuff;
```

改變成

```
uses
  turtlegraphics, ($u apple2:toolstuff.code ) crtstuff;
```

- 增加全盤常數宣告：

• 高等 Pascal 程式設計技巧 •

```
<var>
  sqpic: array [contents] of picture;
  xorg, yorg: integer;

<const>
  XMAX = 279;           { Maximum screen x-coordinate }
  YMAX = 191;           { Maximum screen y-coordinate }
  SQSIZ = 18;           { Board square size in dots }
  PICSIZ = 17;          { Picture size in dots }
  CHARWID = 7;          { Character width in dots }
  ROWSIZE = 4;          { Picture width in bytes }
```

• 增加全盤變數宣告：

```
<type>
  picture = packed array [1..PICSIZ, 1..PICSIZ] of boolean;
```

sqpic 陣列利用顯示方格的內容掌握圖形。**xorg** 和 **yorg** 變數包括圖形螢幕上棋盤的左上角座標。

- 下步驟是增加設定初值程序 **initrev 2** 的代碼，以設定 **sqpic** 布耳陣列，並且設定 **xorg** 和 **yorg**。首先增加常數宣告：

```
<const>
  R1SQ = 54;
  R2SQ = 44;
  R = 9;
```

我們也需要在 **initrev** 內另外宣告二個變數：

```
<var>
  j, i2, r2: integer;
```

最後，下列代碼必須插在 **initrev 2** 中最後的 **end** 敘述之前：

```
xorg := (XMAX - 8 * SQSIZ) div 2;
yorg := (YMAX - 8 * SQSIZ) div 2;
for i := 1 to PICSIZ do begin
  i2 := sqr(i - R);
  for j := 1 to PICSIZ do begin
    r2 := i2 + sqr(j - R);
    sqpic[EMPTY, i, j] := FALSE;
```



```

sqpic[LIGHT, i, j] := (r2 < R1SQ);
sqpic[DARK, i, j] := (r2 < R1SQ) and (r2 > R2SQ);
sqpic[BORDER, i, j] := (r2 < R1SQ) and
((i = R) or (j = R) or (i = j) or (i = PICSIZE + 1 - j))
end
end

```

圖形陣列的初設也許有點混亂。對於每一陣列元素〔i, j〕而言，程式計算出一整數值 **rz**，該值是從陣列的中央〔9, 9〕到該元素距離的平方。

sqpic〔EMPTY〕 布耳陣列的所有元素，均設定成 **FALSE**。簡言之，螢幕上空的方格都填滿成黑色。

如果 **rz** 的值小於 54，**LIGHT** 方格圖形就被打開（設定成 **TRUE**）；否則就被關閉（設定成 **FALSE**）。這造成螢幕上的四周為白色。

同理，**rz** 值介於 45 和 53 之間，圖形中 **DARK** 方格中的點被打開；螢幕上白點的環圍繞黑色的中央，指示一個黑棋。

最後，**BORDER** 是本文版中的移動選擇游標，例如 **rz** 的值小於 54，圖形中的點設定為 **TRUE**，它位居水平線中央的方格，或是垂直線的中央，或是對角線的中央，如此，則有星狀物在圖形螢幕上面顯示。

• 用下列的圖形版本取代 **dispgrid** 的本文版：

```

procedure dispgrid;
{ Display board grid - graphics }

var
    i, x, y, d: integer;

begin { dispgrid }
    d := 8 * SPSIZ;
    turnto(0);
    y := YORG;
    for i := 0 to 8 do begin
        pencolor(NONE);
        moveto(xorg, y);
        pencolor(WHITE);
        move(d);
        y := y + SPSIZ
    end;
    turn(90);
    x := xorg;
    for i := 0 to 8 do begin
        pencolor(NONE);

```

```
        moveto(x, yorg);
        pencolor(WHITE);
        move(d);
        x := x + SCSIZ
    end
end;
```

- 在 **dispgrid** 程序後面，加上以下六個程序。第一個是 **initgraphics**，它把本文顯示螢幕變換成爲圖形螢幕：

```
procedure initgraphics;
( Initialize graphics screen )

begin ( initgraphics )
    initturtle
end;
```

APPLE II 80 行介面卡的使用人，必須加一些代碼，並且輸送一些與硬體相關的命令到介面卡，或由使用人直接執行一些動作。例如 ALS 80 行介面卡，需要額外的

```
write(chr(20), 'A1')
```

包含在程序之內，自介面卡的視頻（ video ）信號，變換成 APPLE 的正常視頻信號。

第二個程序是 **termgraphics**，把圖形螢幕換回本文螢幕：

```
procedure termgraphics;
( Back to text screen )

begin ( termgraphics )
    textmode
end;
```

termgraphics 由 **initgraphics** 廢棄。80 行介面卡的使用戶，必須根據其狀況插入一些代碼。例如，ALS 需要加

```
write(chr(20), 'B1')
```

其餘四個程序，在圖形螢幕上執行命令本文操縱功能。

```

move[ 1] := 33;
move[ 2] := 34;
move[ 3] := 35;
move[ 4] := 36;
move[ 5] := 43;
move[ 6] := 46;
move[ 7] := 53;

move[ 8] := 56;
move[ 9] := 63;
move[10] := 64;
move[11] := 65;
move[12] := 66
end
end;
for i := 1 to 8 do
  for j := 1 to 8 do
    setsquare(10 * i + j, EMPTY);
  setsquare(44, LIGHT);
  setsquare(55, LIGHT);
  setsquare(45, DARK);
  setsquare(54, DARK);
  center('Do you want white or black?(W/B):', ymarg + 18);
  case getkey(ch, ['W', 'B'], TRUE) of
    'W': begin
      human := LIGHT;
      posstr('Computer is black', xmarg, ymarg + 17)
    end;
    'B': begin
      human := DARK;
      posstr('Computer is white', xmarg, ymarg + 17)
    end
  end;
end;
center('Enter lookahead for computer(1-6):', ymarg + 18);
lookahead := ord(getkey(ch, ['1'..'6'], FALSE)) - 48;
eraseline(ymarg + 18);
posstr('Lookahead:', xmarg + 22, ymarg + 17);
write(ch);
posstr('Computer:', xmarg, ymarg + 18);
posstr('Human:', xmarg + 24, ymarg + 18);
computer := other(human)
end;

```

initgame 藉設定邊緣方格為 **BORDER** 和內部方格定為 **EMPTY**，來設定主要棋盤的初值。在棋盤的中央放置二個白棋（**LIGHT**）和二個黑棋（**DARK**）。同時在其四周設定主棋盤的 12 個可能的着手之初值。這些方格是我們為雙方檢查合法着手的對象。（其餘着手不考慮）

定主棋盤初值後，**initgame** 要求使用人選擇顏色，和鍵入預見值。因為使用人的輸入鍵，是透過 **reversi**, **initgame** 使用 **getkey** 來接受的，而不使用 **getstring**。注意參數傳送到 **getkey**，以保證使用人鍵入的回答合於規定。

```
pencolor(NONE);
moveto((XMAX - (length(s) + 1) * CHARWID - 3) div 2, YMAX - 11);
turnto(0);
pencolor(WHITE);
move(wd);
turn(90);
move(ht);
turn(90);
move(wd);
turn(90);
move(ht)
end;
```

- 用本版取代本文版，使用 **drawback** 來顯示我們前面所定義的圖形：

```
procedure dispsquare(k: squarenum; c: contents);
{ Put picture in square on graphics screen }
begin { dispsquare }
    drawblock(sqpic[c], ROWSIZE, 0, 0, PICSIZE, PICSIZE,
              xorg + SQSIZ * (k div 10 - 1) + 1,
              yorg + SQSIZ * (k mod 10 - 1) + 1, 10)
end;
```

- 在 **initgame** 程序中，程式要求在圖形螢幕上顯示資訊，而不在本文螢幕上顯示，有下列六種修飾可完成這個事實：

- (a) 把 **initgame** 的第一行

```
eraseline(ymarg + 17);
```

改變成

```
geraseline(yorg - 10);
```

- (b) 改變

```
center('Do you want white or black?(W/B):', ymarg + 18);
```

• 6 遊戲和戰略 •

成爲

```
gcenter('Do you want white or black?(W/B):', yorg - 20);
```

(c) 改變

```
posstr('Computer is black', xmarg, ymarg + 17)
```

成爲

```
gposstr('Computer is black', 0, yorg - 10)
```

(d) 同理，改變

```
posstr('Computer is white', xmarg, ymarg + 17)
```

成爲

```
gposstr('Computer is white', 0, yorg - 10)
```

(e) 改變

```
center('Enter lookahead for computer(1-6):', ymarg + 18);
```

成爲

```
gcenter('Enter lookahead for computer(1-6):', yorg - 20);
```

(f) 最後改變五行

```
eraseline(ymarg + 18);  
posstr('Lookahead:', xmarg + 22, ymarg + 17  
write(ch);  
posstr('Computer:', xmarg, ymarg + 18);  
posstr('Human:', xmarg + 24, ymarg + 18);
```

成爲

• 高等 Pascal 程式設計技巧 •

```
geraseline(yorg - 20);
gposstr('Lookahead:', XMAX - 11 * CHARWID, yorg - 10);
wchar(ch);
gposstr('Computer:', 0, yorg - 20);
gposstr('Human:', XMAX - 8 * CHARWID, yorg - 20);
```

- **dispscore** 程式也必須修改，把分類顯示在圖形螢幕之上。改變

```
posstr(s, xmargin + 10, ymargin + 18);
```

成為

```
gposstr(s, 9 * CHARWID, yorg - 20);
```

並且把

```
posstr(s, xmargin + 31, ymargin + 18)
```

改變成

```
gposstr(s, XMAX - 2 * CHARWID, yorg - 20)
```

- **declarewinner** 常式必須修飾，改變下列各行

```
center(concat('I won by ', s), ymargin + 17)
...
center(concat('You won by ', s), ymargin + 17)
...
center('We have tied!', ymargin + 17)
```

成為

```
gcenter(concat('I won by ', s), yorg - 10)
```

• 6 遊戲和戰略 •

```
...  
gcenter(concat('You won by ', s), yorg - 10)  
...
```

```
gcenter('We have tied!', yorg - 10)
```

- 最後，把 **reversi** 的主常式中作四種改變：

- a. 改變下二行

```
crt(CLEAR);  
disptitle('reversi');
```

成為

```
initgraphics;  
gdisptitle('reversi');
```

- b. 改變下行

```
center('Play again?(Y/N):', ymag + 18);
```

成為

```
gcenter('Play again?(Y/N):', yorg - 20);
```

- c. 把下行

```
eraseline(ymag + 18);
```

改變成

```
geraseline(yorg - 20);
```

- d. 改變最後的行是

```
crt(CLEAR)
```

改爲

teragraphics

建議 (Suggestion)

改進 **reversi** 是非常困難，首先改進程式的效率是第一要務；一個交作式的遊戲程式都不快。如前面的建議，我們必須考量如何改進收獲最大。

把 **findmax** 和 **findmin** 組合成一個非遞迴函數，是值得研究的，爲了儲存棋盤、和樹狀結構每一階層的移動表列等像堆疊一樣的資料結構都要維護。通常一個非遞迴常式較相等的遞迴常式爲快，雖然增加了效率，更具複雜性。

你也許考慮修飾資料結構，或用組合語言改寫常式，以期改進效率，同時閱讀本章最後的技巧，可能會有所助益。

其他改進途徑是改進程式遊戲的特性。雖然本版的 **reversi** 能打敗新手，却非有經驗的人的對手。最簡單的戰略是改變 **eval** 函數的常數值 **ki**。你也許希望有着手的重要因素，在早期強調機動性和戰略性之值，在快結速時則着重如何才能使棋子增加，當然你也可能發現更好的方法做最佳決定。

爲判斷你對着手方面的改變是否改進，無妨用兩種版本的程式對抗，看看何者獲勝；你也可在若干競賽中決定那一個最好。本挑戰是使競賽盡可能自動化，很容易建立一群參與比賽的人，每人戰略不同。若干小時（日子、甚至星期）之後，找到全面勝利者。

程式的審美觀也要改進，修改程式以便更有效的應用硬體特性，利用音響，圖形，輸入裝置等。有一種值得改進的是，爲現在使用的版本，用某種方法通知使用人，電子計算機最後

着手和被翻的棋子。

增加一個把流程棋盤狀況由列表機印出的命令；如你的列表機有此功能，在高解度上爲之。增加一個命令將使用者的着手各種變化都列印之。讓使用人利用更改參數值 k_i 來改變電子計算的戰略。增加一選擇強迫電子計算機着次佳的棋（或第三好，第四好，等等）。

效率的改進與較高的預見值有關。增加命令把棋賽的部份磁碟載入或存到磁碟上。

讓使用人限制電子計算機使用時間的上限（如你的電子計算機有 real-time 鐘的設備，是很容易辦到的。）如有這種時間限制，如何能調整 minimax 戰略呢？

最後，你也許希望設計其他遊戲。敬祝好運！

推薦閱讀 (Recommended Reading)

J. Scarne 著的 Scarne's Encyclopedia of Games 中描述 Reversi 和其他遊戲。

David Levy's 月刊電子計算機智力遊戲欄是一很好的電子計算機遊戲程式參考資料。本欄於 1980 發行，1982 年三月停刊。1981 年 6 月號刊登 Reversi 部份。

Etudes for Programmers (C. Wetherell 著) 很清楚，正式的描述樹狀遊戲的搜尋與修剪方法。該書尚討論 Kalah 遊戲。Principles of Artificial Intelligence (N. Nilsson 著) 對於人造智慧有詳細的改進程式之概念。

棋賽方面，Sargan : A Computer Chess Program 一書 (Dan and Kathe Spracklen 著)，描寫變棋程式，並有用 Z-80 組合語言的原始代碼。這是一個爲新遊戲而寫的程式原始概念，該雜誌銷路很廣。

• 高等 Pascal 程式設計技巧 •

殼式排序在 Sortlist 中使用到，該演算法逐字抄自 Software Tools in Pascal (B.Kernighan 和 P. J. Plauger 合著)。本書使用的技巧高超。

7

模擬與漫畫

(*Simulation and Animation*)

除了在電動玩具方面的應用之外，電子計算機通常是用在比較嚴肅的主題，多半需要經過某個團體共同討論模擬以應付各種不同的條件。

也許在預測方面共同討論：在某一時段之內，會有什麼樣的事情發生。也許在解說，表達方面，需要模擬：當然，這種模擬，必定是在「現實社會系統的行爲，已證明這個預測」的前提之下的一種正確的模式。最後，也許在實驗方面需要模擬：改變模式的參數（Parameter），並觀察，其結果的行爲，俾瞭解在類似情況下的正常行爲。在現實社會中，經過實驗發現其本身是不可能成功，或花費不貲，或無法實現的事項，則上述的「實驗」觀念，顯得格外有用。

有時候，「真正的、實體的模式」系統，也需要付諸模擬。例如，密西西比河（Mississippi River）和 chesapeake 灣的流水量的問題。其他著名例子，諸如，飛機或汽車在風洞、隧道的測試等。

電子計算機的模擬系統，相當複雜，包括抽象的數學公式和邏輯決定。通常，它分爲「硬體」（hard）科學和社會（social）科學。硬體科學方面指：物理（physics）學、化學（ch

emistry) 、和工程學(**engineering**) 等；軟體科學指：經濟學(**economics**) 、心理學(**psychology**) 和人類學(**anthropology**) 等。

大多數的商業界，利用電子計算機模式(**computer modeling**)去預測，在不同的價格、市場的戰略、或經濟變化之下的企業行爲。國會預算公室(**The Congressional Budget Office**)，利用一個國家經濟的電子計算機模式，去推測失業、通貨膨脹、和經濟成長。核子工程師們，模擬緊急狀況下的不同之放射。生態學家們(**Ecologists**)利用電子計算機模式、預測人口和人類可用的自然資源。

未來的模擬，有其共同特性——簡單化。把現實社會的系統，翻譯成模式，把其複雜久以簡化，不重要的因素可以忽略。例如，電視遊樂器製造廠，可能不把「氣候」的因素，納入其銷售計劃模式。

簡化的方法，尚有「把確定的數值或關係」指派到那些晦暗不清的元素。例如，商業方面的模擬，由於加花 x 元的廣告經費，則促銷增加 y 元，而 y 是一個複雜的 x 數學函數。在這複雜的數學函數之後，尚有無數的問題，諸如，刊登何種廣告、刊在那裡，人們反映如何？等等。

上述的簡化，使得模擬結果，不夠正確。錯誤的判斷，不好的邏輯，都足以使模擬的結果，完全錯誤。例如判斷「天氣不良、持續下雨多日」，會增加『電視遊樂器』的暢銷。總之，設計良好的模擬，對我們的助益才大。

以上，我們討論一般性的模擬。下面我們則舉出實例二則。第一，**bouncer** 例，模擬「盒內的球的行爲」。第二，**issac** 例，模擬「萬有引力下身體的動作」。

這兩則模擬，都是用生動的漫畫圖案顯示，程式是利用 **APPLE II** 設計的，讀者可嘗試在其他電子計算機仿效設計

之。

資料結構——座標和向量 (Data Structure——Coordinates and Vectors)

在 **bouncer** 和 **isaac** 兩程式裡，都需考慮若干事物的位置、速度、和加速度。

首先考慮描述位置。最簡單的表示法，即以數字表之。例如，蒼蠅在西牆 2.72 米、北牆 3.14 米」處的屋子，高 0.58 米。又如，飛機的位置，可由緯度、經度、和海平面的高度表示之。這些描述事物的位置，亦即該事物的座標，這種方法，則稱之為座標系統。

笛卡爾座標 (Cartesian coordinate) 最常用，上述的三個數字，可分別由 X ， Y 和 Z 表之。換言之，可寫成 (X, Y, Z) 的形式。設 X 為到北牆的距離， Y 為與西牆的距離， Z 表示高，則其座標為 $(3.14, 2.72, 0.58)$ 。笛卡爾座標的原點是 $(0, 0, 0)$ 。

X ， Y ， Z 三者形成一個向量，每一個數都是向量的成份 (component)，這個向量為三維向量 (three dimensions)。

由於 CRT 幕是一個 2 度空間 (2 dimensional system) 系統；吾人可以忽略上例的高度，僅用 (X, Y) 來表示之。

至於蒼蠅的速度，也可用向量表示。例如 $(0.11, 0.39, -0.55)$ 則一秒之後，它的位置在 $(3.14+0.11, 2.72+0.39, 0.58-0.55)$ 或是 $(3.25, 3.11, 0.03)$

由於位置向量為二維，速度向量也可把其第二個分量忽略不計。

最後，考慮加速度，也可用向量表示之。

向量經常解釋或「一個固定長度，指向定距離的箭」。向量 (X, Y, Z) 可畫成，自 $(0, 0, 0)$ 指向 (X, Y, Z) 的箭。箭的長度又稱為「向量長度」。則向量長度是一個

數字，而非向量，其值為 $\sqrt{X^2 + Y^2 + Z^2}$ 。速度向量長度，即該事務的速度。

PASCAL對於向量的表示法，非常明確，三維空間可寫成：

```
<type>
  coordinate = (X, Y, Z);
  vector = array [coordinate] of real;
```

蒼蠅的位置、速度和加速度，皆可宣告成向量：

```
<var>
  mosqpos, mosqvel: vector;
```

並且把數字指派 (assign) 給向量的每一分量 (component)：

```
...
mosqpos[X] := 3.14;
mosqpos[Y] := 2.72;
mosqpos[Z] := 0.58;
...
mosqvel[X] := 0.11;
mosqvel[Y] := 0.39;
mosqvel[Z] := -0.55;
...
```

則向量長度的計算，可設計成簡單的 PASCAL FUNCTION 如下：

```
function magnitude(var v: vector): real;
{ Calculate magnitude of a vector }

begin { magnitude }
  magnitude := sqrt(sqr(v[X]) + sqr(v[Y]) + sqr(v[Z]))
end;
```

sqr 和 **sqrt** 分別為標準 PASCAL 常規中的平方和平方根。

爲了方便在螢幕上表達蒼蠅的狀況，吾人無論是 **bouncer** 或 **issac** 皆忽略 Z 座標，則其座標可改爲：

```
<type>  
coordinate = (X, Y);
```

實數 (Real Numbers)

在第三、四章裡，我們不用實數，而用 **fixed** 和 **xyeal** 資料型態，造成溢位 (overflow) 或超下限 (underflow) 的嚴重問題。

若我們設計一套商用模擬程式，使用浮點算術 (floating point arithmetic) 比較安全。

在典型的模擬中，無論是溢位或超下限，都是非常嚴重的。一個模擬程式的結果，造成溢位或超下限的錯誤，通常在其模式本身的流程都會顯示出來。以商業模擬為例，依模式來看，該公司每周花費 10^{40} 元，則一定會導致溢位的錯誤，爲了處理程式，就不能著重在資料型態 (data type) 了，而必須改變模式的假定、與其內部工作。

對於模擬程式而言，利用實數是個有力的引數 (argument)。十進位小數點 (decimal point) 和符號，由語言自動處理之。而內建功能 (built-in) 超越函數 (transcendental functions)，例如 Sin, exp 等等則以實數作爲引數 (arguments)，並把結果 (實數) 傳回主程式。

bouncer 是我們的第一個模擬的例子，它描述在封閉的盒內之一群球的運動。每執行 **bouncer** 一次，使用者就產生模擬的若干的運轉 (runs)，對於每一個運轉 (run)，使用者提供若干個參數 (parameters) 控制運轉的條件。運轉的參數有 4：

- 盒內的球數：每次運轉，都由使用者可以告訴電子計算機，運動的球數為 40，當然，數目可以變動，數目愈大，則程式運轉的就愈慢。球數參數為一整數，以 **nballs** 表之。
- 時間區間：這個參數粗略的控制計算；它以 **delta** 或 **dt** 表之，**dt** 說明，根據位置和速度估計連續兩球的時間。(**bouncer**) 利用瞬間的位置和速度，去計算短時間之後的位置和速度，此後又可反覆作這種推算出新的時間的位置和速度。) 時間間隔太大，準確性就變小，程式速度即減慢。通常時間區間介於 0.0001 和 1.0 之間。
- 加速度向量：各球的加速度，可以設定為一常數 (或為 0)，加速度用來表示地心引力，它是一個向量，所以兩個數分別代表 X 方和 Y 方向的加速度。這兩個分量，可由用者轉入，其值介於 -1.0 和 1.0 之間。在 **bouncer** 中，這加速度參數是 **acc**，也是一個向量型態 (**vector type**) 的一個變數。
- 顯示球的途徑與否：球的途徑可以顯示，也可不顯示，由 **bouncer** 選擇。該參數用布耳變數 (**boolean variable**) **Showpath** 表示。

bouncer 例子的球在盒內最初的位置，是隨機 (**random**) 的、初速度亦然。這兩個值由使用者鍵入。

鍵入運轉變數，**bouncer** 即在付予的條件下，圖示球的運動。圖 7 - 1 示三個球的快射的運轉；X - 加速度為 0.0，Y

• 7 模擬與漫畫 •

— 加速度爲 -1.0 ，時間區間 0.1 。

模擬程式不斷的運轉，直到使用者按下任何鍵；它就回到本文 (text)，並要球使用者賦允新的參數值。想要結束本程式，鍵入 0。

茲列舉 **bouncer** 的主常式 (main routine)：

```
($s+)
program bouncer;
{ Bouncing ball simulation }

uses
  applestuff,           { for note, random, and keypress }
  turtlegraphics,       { for graphics routines }
  ($u apple2:toolstuff.code)
  crtstuff;

const
  MAXBALLS = 40;         { Maximum # of balls in box }
  PICSIZE = 5;           { Picture size in dots }

type
  coordinate = (X, Y);
  vector = array [coordinate] of real;
  picture = packed array [1..PICSIZE, 1..PICSIZE] of boolean;

var
  ballpic: picture;
  i, margin, nballs: integer;
  showpath: boolean;
  acc: vector;
  dt: real;
  prompt: array [1..5] of string;

{-----}
{ Modules to be inserted here: }
{      initbounce      }
{      getbparams      }
{      runbouncer      }
{-----}

begin { bouncer }
  crt(CLEAR);
  disptitle('bouncer');
  initbounce;
  for i := 1 to 5 do
    posstr(prompt[i], margin, 8 + 2 * i);
  repeat
    getbparams;
    if nballs > 0 then
      runbouncer
  until nballs <= 0;
  crt(CLEAR)
end.
```

```
procedure initbounce;  
( Initialize bouncer globals )  
  
var  
    i, j: integer;  
  
begin ( initbounce )  
    prompt[1] := 'Number of balls (0 to exit):';  
    prompt[2] := 'Time interval:';  
    prompt[3] := 'X-acceleration(-1.0 - +1.0):';  
    prompt[4] := 'Y-acceleration(-1.0 - +1.0):';  
    prompt[5] := 'Show ball paths(Y/N):';  
    margin := (MAXCRTCOL - 29) div 2;  
    for i := 1 to 5 do  
        for j := 1 to 5 do  
            ballpic[i, j] := ((i in [1, 5]) and (j in [2..4])) or  
                               ((j in [1, 5]) and (i in [2..4]))  
        end;  
    end;  
end;
```

注意，**bouncer**裡的圖形資料型態 (**picture data type**) 同第六章的 **reversi**，唯一的差別在圖形陣列 (**array**) 的大小，該值用 **PICSIZE** 常數表之，在 **reversi** 中其值 17，本例裡該值為 5。

全盤變數定初值 (Initializing Global Variables)

習慣上，我們以分開的常式去定全盤變數的初值：

initbounce 將下列變數定初值：

- 提示列陣 (**Prompt**)：該列陣包含五個字串，用來作前面描述的運轉參數之提示用。
- 邊際 (**Margin**)：控制 幕顯示 **Prompt** 的水平位置 (確保 40 行或 80 行 幕都能正常的顯示)。
- 圖形列陣 **ballpic**：顯示 幕上的球。經定初值之後，如列陣的元素為真 **TRUE**，幕上對應的點是亮的，其為假 (**FALSE**)，則對應的點呈暗。

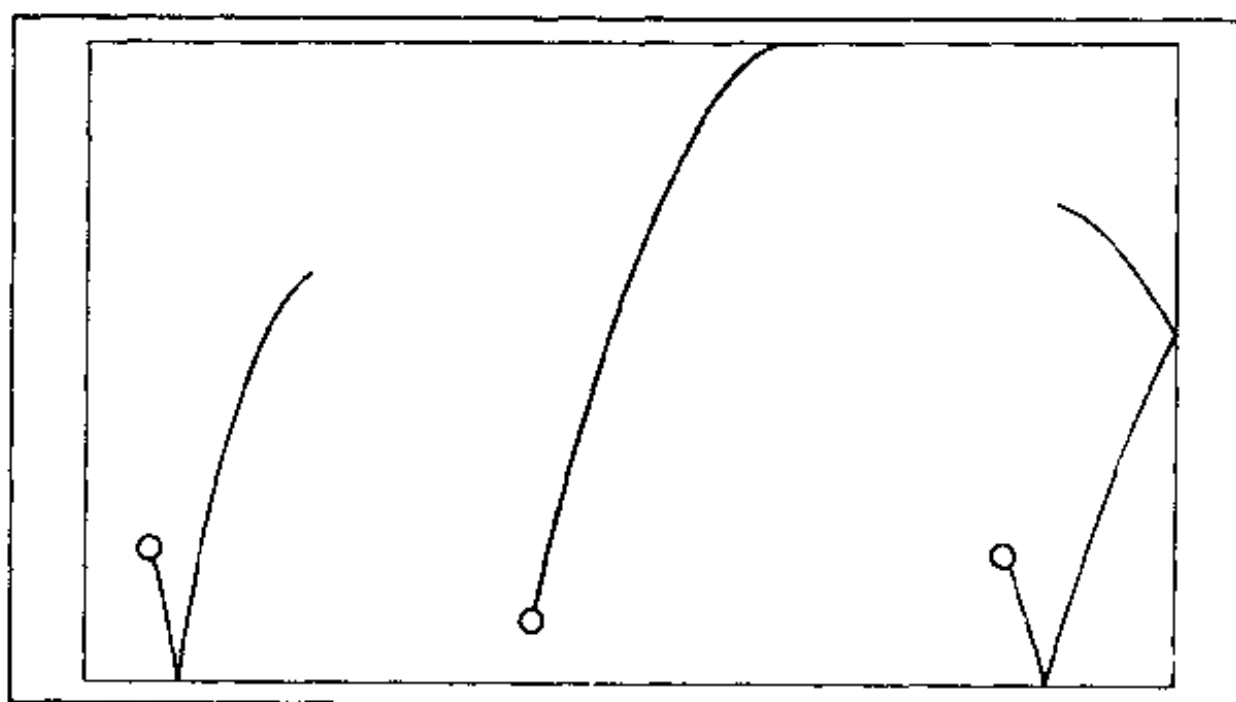


圖 7-1 bouncer 運轉

取得參數常式 (Getbparams)

getbparams 常式是一個簡單的交談式的輸入常式，本常式並無新奇之處，設計方面也不困難。

茲列舉 **getbparams** 如下：

```

procedure getbparams,
( Get parameters for bouncer run )

type
    format = (FIXEDPOINT, SCIENTIFIC);

(-----)
( Modules to be included here: )
(      stoi      )
(      itos      )
(      getint     )
(      getreal    )
(-----)

begin ( getbparams )
    nballs := getint(margin + length(prompt[1]), 10, 0, MAXBALLS, 0, FALSE);
    if nballs > 0 then begin
        dt := getreal(margin + length(prompt[2]), 12, 0.0001, 1.0, 0.0,
            FIXEDPOINT, 6, 4, FALSE);
        acc[X] := getreal(margin + length(prompt[3]), 14, -1.0, 1.0, 0.0,
            FIXEDPOINT, 6, 2, TRUE);
        acc[Y] := getreal(margin + length(prompt[4]), 16, -1.0, 1.0, 0.0,
            FIXEDPOINT, 6, 2, TRUE);
        showpath := getboolean(margin + length(prompt[5]), 18, FALSE, FALSE);
    end
end;

```

整數輸入 (Integer Input)

讀者可回憶一下，第五章排印程式 `print` 的整數——輸入常式，該常式利用 `getfixed` 自使用者取得固定資料型態，再將該值轉換成一個整數。現在，我們另外設計一個 `getint` 函數，直接使用整數。

`getint` 函數的邏輯與其他交談式輸入常規一樣，呼叫形式與第五章相同：

```
function getint(col,row,mini,maxi,defi:integer; defaulted:boolean):integer;
( Get integer from user )

var
  mins, maxs, defs, s: string;
  maxlen, i, int: integer;
  booboo, good: boolean;

begin ( getint )
  itos(mini, 0, mins);
  itos(maxi, 0, maxs);
  if length(mins) > length(maxs) then
    maxlen := length(mins)
  else
    maxlen := length(maxs);
  if defaulted then
    itos(defi, 0, defs)
  else
    defs := '';
  booboo := FALSE;
  repeat
    getstring(s, maxlen, col, row, defs, ['0'..'9', '+', '-'], FALSE);
    good := (length(s) > 0);
    if good then begin
      i := 1;
      good := stoi(s, i, int)
    end;
    if good then ( still ok )
      good := (int >= mini) and (int <= maxi);
    if good then begin
      itos(int, maxlen, s);
      posstr(s, col, row)
    end
    else begin
      booboo := TRUE;
      crt(BEEP);
      center('Please enter a number between', MAXCRTROW - 1);
      center(concat(mins, ' and ', maxs), MAXCRTROW)
    end
  until good;
  if booboo then begin
```

```

    gotoxy(0, MAXCROW - 1);
    crt(ERASEOS)
end;
getint := int
end;

```

getint 利用 **itos** 常式（詳見第六章）把一個整數轉換成爲一個字串，我們也希望設計一個把字串轉換成一個整數。

在第三章裡有個 **stof** 函數：

```
good := stoi(s, i, int);
```

這函數在 **s** 字串中找第 **i** 個字元，整數則由參數 **int** 傳回，參數 **i** 指向字串字元，這個字元若非數字或字串的結尾就停下來。**stoi** 以一個布爾數值回轉給 **good**，若是 **TRUE** 則表示這個字串代表一有效整數，若是 **FALSE**，則字串包含一個大於 **MAXINT** 的整數，或小於 **MAXINT** 整數。

stoi 函數如下：

```

function stoi(var s: string; var i, int: integer): boolean;
{ Convert string to integer }

const
    MDIV10 = 3276;      { MAXINT div 10 }
    MMOD10 = 7;         { MAXINT mod 10 }
var
    ok, negnum: boolean;
    digit: integer;
    c: char;
begin { stoi }
    int := 0;
    ok := TRUE;
    negnum := FALSE;
    if gnbchar(s, i, c) in ['+', '-'] then begin
        negnum := (c = '-');
        i := i + 1
    end;
    while gnbchar(s, i, c) in ['0'..'9'] do begin
        if ok then begin
            digit := ord(c) - 48;
            if (int > MDIV10) or ((int = MDIV10) and (digit > MMOD10)) then
                ok := FALSE
            else
                int := 10 * int + digit
        end
    end;
end;

```

```
end;  
i := i + 1  
end;  
if negnum then  
  int := -int;  
stoi := ok  
end;
```

MDIV 10 設定成 $\text{MAXINT} \div 10$ 而 MOD 10 設定成 MAXINT。我們必須保證

int := 10 * int + digit

不會溢位，這個 MAXINT 的上限是 32767；其他版本的 PASCAL 也許不同。

stoi 和 itos 可用來取代 stof 和 fto

實數輸入 (Real Nambre Input)

下個步驟是設計一個常式 **getreal**，從使用者那裡取得一個實數。

就如前面提到的，由於 PASCAL 有不同的版本，內建實數資料型態的準確度，範圍和儲存記憶位置不同。甚而有的 PASCAL 允許程式設計師，選擇最適合的實數資料型態，而且不只一個。爲了寫一個值得信賴的交談式的輸入實數的工具，有必要知道其在實數方面的限制，這個資訊有助於實施的文件 (implementations documentation)。

APPLE PASCAL 的實數資料型態，大約 7.2 十進數位，精確度是 24 位元 (24 bits)，正實數範圍介乎 2^{126} 和 2^{-126} 之間，每一個實數變數占用 4 位元組 (4 bytes)，這些資料

都是我們設計一個常式，去作字串和實數互相轉換時應該考慮到的，才能避免發生溢位和超下限的錯誤。

getreal 函數利用 **getstring** 去接受使用者的輸入，並作為字串變數，進而將字串轉換成所希望的資料型態 **getreal** 可用下列方式呼叫：

```
r := getreal(col, row, minr, maxr, defr, fmt, width, ndigs,
                                     defaulted);
```

各個呼叫參數含義分述如下：

- **col** 和 **row** 說明在 **CRT** 螢幕輸入位置。
- **minr** 和 **maxr** 是輸入數目的下限與上限，若輸入的數目不合要求，**getreal** 即在螢幕的下方，給一個錯誤的訊息。
- **defr** 為缺設回答 (default answer) 它只限在布耳參數被缺設為 **TRUE** 時才用得到，否則不提供缺設置。
- **fmt** 為控制數目的格式 (format)，它和第 4 章內 **FIXEDPOINT** 或 **SCIENTIFIC** 的定義一樣。
- **Width** 控制輸入欄位的長度，鍵入數字則使用者可使用最大寬度的字元。
- **ndigs** 是數目的位數。

若使用者打入的數目，經核對為有效，於是返回呼叫者，作為函數的結果。

getreal 的邏輯與第 3 章的 **getfixed** 很像，茲列舉如下：

```
function getreal(col: integer; minr, maxr, defreal: real;
                 fmt: format; width, ndigs: integer; defaulted: boolean): real;
{ Get real number from user }

type
  calcstatus = (OK, OVERFLOW, UNDERFLOW, ZERODIVIDE);
```

```

var
  s, mins, maxs, defs: string;
  good, booboo: boolean;
  i: integer;
  r: real;
  okset: charset;

{-----}
{ Modules to be included here: }
{      stor      }
{      rtos      }
{-----}

begin { getreal }
  okset := ['0'..'9', '.', '+', '-', 'E', 'e'];
  rtos(minr, fmt, 0, ndigs, mins);
  rtos(maxr, fmt, 0, ndigs, maxs);
  if defaulted then
    rtos(defreal, fmt, 0, ndigs, defs)
  else
    defs := '';
  booboo := FALSE;
  repeat
    getstring(s, width, col, row, defs, okset, FALSE);
    good := (length(s) > 0);
    if good then begin
      i := 1;
      good := (stor(s, i, r) = OK)
    end;
    if good then
      good := (r >= minr) and (r <= maxr);
    if good then begin
      rtos(r, fmt, width, ndigs, s);
      posstr(s, col, row)
    end
    else begin
      booboo := TRUE;
      crt(BEEP);
      center('Please enter a number between', MAXCRTROW - 1);
      center(concat(mins, ' and ', maxs), MAXCRTROW)
    end
  until good;
  if booboo then begin
    gotoxy(0, MAXCRTROW - 1);
    crt(ERASEOS)
  end;
  getreal := r
end;

```

正如前面 `get <type>` 常式一樣，`getreal` 需要一個分開的常規，作字串與所需的資料型態相互轉換。

首先考慮 `rtos` 常式，根據格式，把實數翻譯成一個字串，令 `r` 表實數，`s` 表字串，呼叫方式如下：

```
rtos(r, fmt, width, ndigs, s);
```


如參數 **fmt** 是定點 (FIXEDPOINT)，字串的格式亦同。若是用科學的記法，則 **fmt** 為 SCIENTIFIC。無論是那一種情況，在十進位小數點之後顯示 **ndigs** 個數字，而字串在左邊補上空白，必要時字串長度設定為 **width** 個字元，若比 **width** 字串長，就製成更長的字串。

rtos 與 **fmt** 的值有關，該程序設計如下：

```

procedure rtos(r: real; fmt: format; width, ndigs: integer; var s: string);
{ Convert real to string }

{-----}
{ Modules to be included here: }
{      rtofix      }
{      rtosci      }
{      rightjust   }
{-----}

begin { rtos }
  case fmt of
    FIXEDPOINT:
      rtofix(r, ndigs, s);
    SCIENTIFIC:
      rtosci(r, ndigs, s);
  end;
  rightjust(s, width);
end;
```

rtofix 程序 (Procedure) 把一個實數翻譯成一個字串，其格式為 FIXEDPOINT：

若 **r** 經轉換成爲一個負數；字串 **s** 就被設定「-」，而 **r** 設定成 -**r**；否則，**s** 被設定成空串 (null string)，而 **r** 不變。

此後，**r** 被捨棄成 **ndigs**，在小數點後再加一個捨棄因素 (rounding factor)，如 **ndigit=0**，rounding factor = 0.5；如 **ndigit=1**，rounding factor = 0.05 等等。**rtofix** 核對 **ndigs** 之值，以確保不致於產生超下限的錯誤。使用

```
procedure rtofix(r: real; ndigs: integer; var s: string);
{ Convert real to string, fixed point format }

const
  MAXPWR = 37;           { Maximum # of digits to round }

var
  i, digit, nbefore: integer;

begin { rtofix }
  if r < 0.0 then begin
    s := '-';
    r := -r
  end
  else
    s := '';
  if (ndigs >= 0) and (ndigs <= MAXPWR) then
    r := r + 0.5/pwr10en(ndigs);
  nbefore := 1;
  while r >= 10.0 do begin
    r := r/10.0;
    nbefore := nbefore + 1
  end;
  for i := 1 to nbefore + ndigs do begin
    if i = nbefore + 1 then
      addchar(s, '.', MAXSTR);
    digit := trunc(r);
    addchar(s, chr(48 + digit), MAXSTR);
    r := 10.0 * (r - digit)
  end
end;
```

APPLE PASCAL'S 的系統，在計算 rounding factor 時，**ndigs** 大於 37 就會產生錯誤。

經過捨棄之後，**rtofix** 就根據連續除以 10 的方式去計數位 (**digit**)，直到其小於 10 為止，是以 **r** 的範圍才能確保為：

$$0.0 \leq r < 10.0$$

nbefore 為十進位小數點前的數位的個數，注意，其最小為一，那怕十進位小數點為 0，其個數仍為 1。

是故，輸出的字串包括 **nbefore + ndigs** 個位數。而 for 這個環部中，每次處理一個 **r** 的前導位數 (leading digit)

，將之轉換成字串，並加儲於字串 *s* 的結尾，如是則十進位小數點一定可以放到正確的位置。

rtofix 利用 APPLE PASCAL (LICSD PASCAL 亦然) 的內建函數 **Pwroften** 計算 rounding factor :

```
x := pwroften(n)
```

該函數把 *x* 設定為 10 之第 *n* 次冪 (power) , 參數 *n* 必須在 0 到 38 之間 (APPLE PASCAL) , 詳見附錄 A 。

若 **rtofix** 使用不當時，在輸出字串中亦會造成無意義的字串，例如 *r* = 11.0 而 *ndigs* = 10，則輸出字串可能變成 11.00002384 。

這個小錯誤，可以更改 **rtofcx**，讓他不再發生，可以在十進位小數點的左邊捨棄之，特別是 *ndigs* 為負數時採用之，例如 *ndigs* = -2，則被捨去之數近乎 100 倍。

rtosci 相當容易設計，吾人希望 **rtosci** 回復一個字串給 **rtofix**。形式如下：

<任選之符號> <小數部份> e <指數部份>

而小數部份的範圍是：

$$1.0 \leq \text{<小數部份>} < 10.0$$

若輸入的實數不等於 0，則指數部份是一個整數。

根據上述方式，輸入數目 *r*，可分為小數部份 **frac**，和指數部份 **expo**，再分別轉換成字串。首先，定 **frac** 為 *r* 而 **expo** 為 0，如 **frac** 不小於 10，則不斷地除以 10，直到它小於 10 為止，每除以 10 一次，**expo** 就遞增之。例如，輸入 4892.45，處理完畢之後，**frac** = 4.89245，而 **expo** = 3。

換言之，**frac** 初值小於 1.0 (但不為 0)，則連續乘以 10

，直到它大於或等於 1.0 為止，每乘以 10 次，**expo** 就遞減 1，如此 $r = 0.0423$ ，最後 $\text{frac} = 4.23$ 而 $\text{expo} = -2$ 。

```
procedure rtosci(r: real; ndigs: integer; var s: string);
( Convert real to string, scientific notation )

var
  frac: real;
  expo: integer;
  s1, s2: string;

begin ( rtosci )
  frac := r;
  expo := 0;
  if abs(frac) >= 10.0 then
    repeat
      frac := frac/10.0;
      expo := expo + 1
    until abs(frac) < 10.0
  else if (abs(frac) < 1.0) and (r <> 0.0) then
    repeat
      frac := 10.0 * frac;
      expo := expo - 1
    until abs(frac) >= 1.0;
  rtofix(frac, ndigs, s1);
  itos(expo, 0, s2);
  s := concat(s1, 'e', s2)
end;
```

注意， $r = \text{負數}$ 或 $r = 0$ ，**rtosci** 都可以很正確的處理。此外，**rtos** 尚需調整右方，其處理方式與調整左方時類似。該程序設計如下：

```
procedure rightjust(var s: string; width: integer);
( Pad string with blanks on left to expand to specified width )

var
  nblanks: integer;

begin ( rightjust )
  nblanks := width - length(s);
  if nblanks > 0 then begin
    ($r-)
    moveright(s[1], s[nblanks + 1], length(s));
    fillchar(s[1], nblanks, ' ');
    s[0] := chr(width)
    ($r+)
  end
end;
```

這常式先得計算，在字串的右方要補多少個空白才能使長

度變成 **width** 個字元，它補空白的方法，是透過 **moveright** 常式，這與 **moveright** 處理方式類似，詳見第五章。呼叫：

```
moveright(source, destination, nbytes)
```

自 **source** 的位置搬移到 **soarce** 和 **destination** 計 **nbytes** 個位元組。在搬移時要注意到 **source** 和 **destination** 有無重疊之處。

搬移完成之後，利用 **APPLE** 和 **UCSD PASCAL** 的內建常式 **fillchar**（見第五章），把 **nblanks** 個空白加到字串的前方。最後，字串的長度已改變成新的字串長度了。

Stor 常式

stor 是我們要設計的最後一個交談式的實數輸入常式，本常式把字串轉換成實數，設計時要留意溢位和超下限的錯誤。

stor 常式與 **stox** 常式類似（見第四章），其呼叫形式如下：

```
status := stor(s, i, r);
```

注意看數目字串 **s** 的第 **i** 個字元，輸出時，**r** 在字串 **s** 中抽取 **i** 個字串，**stor** 以 **calstatus** 作為結果，回轉給 **status**。若轉換正確，則 **calstatus** 為 **OK**，應則為 **OVERFLOW** 或 **UNDERFLOW**。

仔細讀第四章，有助於設計字串翻譯常式，因為 **stor**，在第四章就用「數」的語法圖解表之（“number” syntax diagram），分辨實數和 **xreal** 字串的重要性，遠不如由程式的用戶來分辨之。

通常最大的問題在於控制的複雜性，解決之道有二：

(\Rightarrow)有意義的數位的數目 **nsig** , 在後繼代碼中。(\Rightarrow) **nafter** 在後繼代碼 (code) 裡。

當數位在字串中被掃描到, 即據之計算 **frac** (小數部份)。
在字串內發現「E」, 即以 **stoi** 將之轉換成指數部份 **expo** ;
若什麼也沒發現, **expo** 即設為 0 , 整個字串掃描完了, 就以 **nsig** 和 **nafter** 為基礎分別調整 **expo** , 然後把 **frac** 和 **expo** 重新組合成實數輸出之。

stor 的虛擬碼 (pseudo-code) 如下:

```
begin
  initialize
  get optional leading sign
  skip over leading non-significant zeros (if any)
  get (zero or more) digits, accumulate fraction
    - accumulate fraction
    - keep track of significant digits seen
  if there's a decimal point
    skip over it
    if no significant digits seen yet
      skip over zeros
      - keep track of # of digits after decimal
    get (zero or more) significant digits after decimal
                                                    point
    - continue to accumulate fraction
    - keep track of # of digits after decimal
    - continue to count significant digits
  if "E" or "e" is seen
    skip over it
    get exponent
  combine fraction and exponent into real
end
```

改成 PASCAL 代碼如下:

```
function stor(var s: string; var i: integer; var r: real): calcstatus;
{ Convert string to real }

const
  MAXRSIG = 10;           { Maximum significant digits in real }
  MAXEXP = 38;            { Maximum real's exponent }
  MAXFRAC = 3.4;          { Maximum real's fraction (approx.) }
  MINEXP = -38;           { Minimum real's exponent }
  MINFRAC = 1.2;          { Minimum real's fraction (approx.) }
```

```

var
  negnum: boolean;
  c: char;
  frac, p10: real;
  expo, nsig, nafter: integer;
  status: calcstatus;

begin ( stor )
  negnum := FALSE;
  frac := 0.0;
  p10 := 1.0;
  expo := 0;
  nsig := 0;
  nafter := 0;
  status := OK;
  if gnbchar(s, i, c) in ['+', '-'] then begin
    negnum := (c = '-');
    i := i + 1;
  end;
  while gnbchar(s, i, c) = '0' do
    i := i + 1;
  while gnbchar(s, i, c) in ['0'..'9'] do begin
    if nsig < MAXRSIG then begin
      frac := frac + p10 * (ord(c) - 48);
      p10 := p10/10.0;
    end;
    nsig := nsig + 1;
    i := i + 1;
  end;
  if c = '.' then begin
    i := i + 1;
    if nsig = 0 then
      while gnbchar(s, i, c) = '0' do begin
        i := i + 1;
        nafter := nafter + 1;
      end;
    while gnbchar(s, i, c) in ['0'..'9'] do begin
      if nsig < MAXRSIG then begin
        frac := frac + p10 * (ord(c) - 48);
        p10 := p10/10.0;
      end;
      nsig := nsig + 1;
      nafter := nafter + 1;
      i := i + 1;
    end;
  end;
  if c in ['E', 'e'] then begin
    i := i + 1;
    if not stoi(s, i, expo) then
      status := OVERFLOW;
  end;
  if status = OK then begin
    expo := expo + nsig - nafter - 1;
    if (expo > MAXEXP) or ((expo = MAXEXP) and (frac > MAXFRAC)) then
      status := OVERFLOW;
    else if (expo < MINEXP) or ((expo = MINEXP) and (frac < MINFRAC)) then
      status := UNDERFLOW;
    else if expo >= 0 then
      r := frac * pwoften(expo);
    else
      r := frac/pwoften(-expo);
  end;
  if negnum then
    r := -r;
  stor := status;
end;

```

（欲證明或了解本常式的最佳方法，就是用手去測試一、二個例子。）

在字串裡已看到有意義數位的一確定數，而此一確定的數目，又與實數資料型態的精密度有關。例入，如輸入字串是「35.00000000383」，最後由 **stor** 回轉它的值的時侯，「383」已不重要了。

MAXRSIG 有意義的數位被轉換之後，**stor** 則越過有意義的數位。**MAXRSIG** 這個常數，被設定大得可以忽略那些附加的數位 (additional digits)；假設，一實數精度為 n 數位，則設定 **MAXRSIG** 為等於或大於 $n + 2$ 的最大整數，比較安全。在前面的代碼，令 **MAXRSIG** 為 10。

也許，你希望重新考慮輸入字串的邏輯是否會使 **stor** 的值發生錯誤，有個有效的方法去保證該常式可以信賴，就值得修定 **stor** 嗎？（注意：**MINFRAC** 和 **MAXFRAC** 不過是近似值）如不提示你，**stor** 中其他錯誤，你能解法嗎？

runbouncer常式

下個步驟是撰寫 **runbouncer** 常式，該常式執行模擬的每一個運轉。

位置和速度都是向量，**PASCAC** 以記錄 (record) 來定義其資料結構 (data structure)，而每一記錄都在一陣列之內：

```
<type>
  ballrec = record
    pos, vel: vector
  end;

<var>
  ball: array [1..MAXBALLS] of ballrec;
```

球數 S 的位置的 x 分量為 **ball**[S]. **pos**[x]，而球

數 10 的速度之 Y 分量是 **ball[10] . vel[Y]**

```

procedure runbouncer;
(Run bouncer simulation )

const
    XMAX = 279;           (Graphics screen maximum x-coordinate )
    YMAX = 191;           (Graphics screen maximum y-coordinate )

type
    ballrec = record
        pos, vel: vector
    end;

var
    ball: array [1..MAXBALLS] of ballrec;
    max, min: vector;
    ch: char;

{-----}
{ Modules to be inserted here: }
{      initgraphics           }
{      plotball               }
{      initbrun               }
{      moveballs              }
{      termgraphics           }
{-----}

begin (runbouncer )
    initgraphics;
    initbrun;
    repeat
        moveballs
    until keypress;
    ch := getkey(ch, [chr(0)..chr(127)], FALSE);
    termgraphics
end;

```

runbouncer 首先呼叫 **initgraphics** , 以準備讓我們使用螢幕顯示圖案, 其次呼叫 **initbrun** 以定運轉的初值。再呼 **moveballs** , 根據初值條件、參數、和前提, 以便在螢幕上搬移球。

呼叫 **moveballs** , **runbouncer** 之後, 立刻核對, 看看使用者按下的鍵。當使用者按一個鍵, **APPLE PASCAL** 的標準單位 (standar unit) **applestuff . keypress** 提供一個布耳函數 (boolean function) **keypress** , 該函數值就為 **TRUE** , 否則函數值為 **FALSE** 。**keypress** 只告訴我們那一個字元正等律去讀, 它並不等我們去按鍵。

keypress 對於中斷程式特別有用，對一個不提供 **keypress** 的 PASCAL 而言，要設計它，就必須具備「電子計算機的作業系統」或「硬體的輸入／輸出」的知識。

當 **runbouncer** 偵測到按下的鍵，它利用 **getkey** 常式去讀該字元的型態；否則，它就等待使用者按鍵，最後，藉 **termgraphics** 回到本文 (text)。

initgraphics 和 **termgraphics** 均於第六章詳述，此處不再重覆。

initbrun 常式

```
procedure initbrun;
( Initialize bouncer ball positions and velocities, box boundaries )

const
  VMAX = 5;                ( Maximum abs. val. for ball velocity component )

var
  a: integer;

(-----)
( Modules to be inserted here: )
(      frame                  )
(      randreal               )
(-----)

begin ( initbrun )
  frame;
  min[X] := 2.0;
  max[X] := XMAX - 2.0;
  min[Y] := 2.0;
  max[Y] := YMAX - 2.0;
  for i := 1 to nballs do
    with ball[i] do begin
      pos[X] := randreal(min[X], max[X]);
      pos[Y] := randreal(min[Y], max[Y]);
      plotball(pos);
      vel[X] := randreal(-VMAX, +VMAX);
      vel[Y] := randreal(-VMAX, +VMAX);
    end
  end;
end;
```

initbrun 常式設定運轉用的一群變數，它並顯示盒子裡每一個球在圖示狀態時的各個引數 (argument)。

initbrun 呼叫 **frame** 常式 (見第六章) 來畫盒子的四周。假定盒子占據整個螢幕。**min** 和 **max** 代表盒子的牆, 球的 **x** 一位置之下限 **min[x]**, 上限是 **max[x]**, **Y** —— 位置亦同。所謂珠的位置就是珠的中心點之位置, 每個球都有「半徑」, 每一半徑皆為二個螢幕單位 (每一點為一單位), 而 **min** 和 **max** 距盒子的牆都是二個單位。

每個球都由 **initbrun** 常式指定一個隨機 (**random**) 位置給它。換言之, 其 **x** —— 座標介於 **min[x]** 和 **max[x]** 之間, **Y** 座標介於 **min[Y]** 和 **max[Y]** 之間。同理, 每一球的初速度即介於 **-VMAX** 和 **+VMAX** 間, **VMAX** 是一常數, 此時 **VMAX** 取值 5.0, 最後, **initbrun** 呼叫 **plotball** 常式, 以顯示各球的最初位置。

隨機實數由 **randreal** 函數提供:

```
function randreal(a, b: real): real;  
( Return random real between specified limits )  
  
begin ( randreal )  
  randreal := a + (b - a) * random/32767.0  
end;
```

這函數在第二章已介紹過, 隨機實數值的範圍是 0 到 32767。

Plotball 常式如下:

```
procedure plotball(pos: vector);  
( Draw or erase ball on screen )  
  
const  
  ROWSIZE = 2;          ( 2 * ((PICSIZE + 15) div 16) - picture size in bytes )  
  
begin ( plotball )  
  drawblock(ballpic, ROWSIZE, 0, 0, PICSIZE, PICSIZE, round(pos[X] - 2.0),  
                                                    round(pos[Y] - 2.0), 6)  
end;
```

Plotball 呼叫 **drawblock** 常式, 以顯示前面提到的在螢

幕上所定義 **ballpic** (圖形陣列)。有兩點值得注意的：(一)、**drawblock** 常式必須由使用者指定圖形陣列的最左下角的 X 和 Y 座標。球的中心點位置是 **POS** 向量，所以在把值傳給 **drawblock** 常式之前，先自 **POS** 的每一分量減 2.0 並把最近的整數捨掉。(二)、我們不僅用 **Plotball** 常式在特定位置上畫一個球，而且把前一次在特定位置畫的球塗掉。我們選擇值為 6 的互斥或算子 (**exclusive-or**) 模式，來搬移圖形陣列的元素到螢幕上面。當檢查到圖形陣列，和其螢幕所對應的點時，若點是 ON 的狀態或陣列為 **TRUE** (兩者不并存)，則螢幕就出現點，否則該點被關掉。

如果螢幕是空的 (所有點 off)，圖形陣列僅複製 (拷貝) 到螢幕上。若螢幕已有圖形陣列的拷貝了，則螢幕點就被關閉。螢幕最初並不是空的。

moveballs 常式

moveballs 常式，對於已知的位置 **Pos** 和速度 **Vel** 和加速度 **acc** 加以計算，並求 **dt** 時間之後，該球的位置、速度、和加速度。所以要把舊位置的圖形抹掉，畫上新的位置。

要簡化上述的大前提，第一，先假定盒子裡的球彼此互相無作用，就像幽靈一樣，小得彼此不會碰撞，這樣可以節省時間，不必去計算每一時隔內，每一對球的距離。

第二，控制球，了解它如何跳開牆壁，只考慮自盒底彈回，假定它對 X 一速度不受影響，只有 Y 一分量的速度，其向上、向下的速度一樣。

最基本的物理知識，就是可讓我們了解，去計算出每個球的新位置、速度、和加速度 (根據初速度、原始位置、最初加速度)。每個球的運動均與其他的球無關，所以計算 X 分量的

時候，Y分量可以忽略。

假定，在某一時間， $P(t)$ 是一個球的位置向量的某一分量。 $t + dt$ 時，新位置分量，就可以下列式子表示之：

$$p(t + dt) := p(t) + dt * \langle \text{average velocity between } t \text{ and } t + dt \rangle$$

我們以一個「虛擬 PASCAL」符號，來代表數學公式。

我們可以拿某間隔時間的平均速度簡化問題，平均速度估計如下：

$$\langle \text{average velocity between } t \text{ and } t + dt \rangle := v(t + dt/2)$$

在間隔時間內加速度為一常數時，上列估計就很精確，否則就失準了。因為我們有一個常數加速度，新舊速度關係如下：

$$v(t + dt/2) := v(t) + a * dt/2$$

此處 a 是加速度，而新位置就變成：

$$p(t + dt) := p(t) + dt * (v(t) + a * dt/2)$$

所以 $t + dt$ 時的速度是：

$$v(t + dt) := v(t) + a * dt$$

記住，我們利用速度、位置、和加速度的計算過程中，只用到一個分量，所以要計算兩次，該計算以 `moveballs` 常式擔任之：

```
procedure moveballs;  
( Move balls on screen )  
  
var  
  i: integer;  
  j: coordinate;  
  newpos: vector;
```

```
{-----}
{ Modules to be inserted here: }
{      line      }
{-----}

begin ( moveballs )
  for i := 1 to nballs do
    with ball[i] do begin
      for j := X to Y do begin
        newpos[j] := pos[j] + dt * (vel[j] + acc[j] * dt/2.0);
        if (newpos[j] < min[j]) or (newpos[j] > max[j]) then begin
          note(2, 2);
          newpos[j] := pos[j];
          vel[j] := -vel[j];
        end
        else
          vel[j] := vel[j] + acc[j] * dt;
        end;
        plotball(pos);
        if showpath then
          line(pos, newpos);
        plotball(newpos);
        pos := newpos;
      end
    end;
end;
```

要留心計算的次序，第一求新位置、若位置的值，小於 **min** 或大於 **max**，則令該球回到原來的位置，並改變其速度分量的正負符號，雖然，此時該球不會碰壁、却能使速度分量能正常的計算。

在 X 分量和 Y 分量計算完畢之後，把原來的圖形塗掉，畫上新位置的圖形。假如 **showpath** 參數是 **TRUE** 時，就在新、舊位置之間畫一條線。最後，**pos** 更新 **newpos** 之後，再度呼叫 **moveball** 程序，以便求下一新位置。

當球跳離牆壁之際，**moveballs** 程序裡就呼叫 **note** 程序，產生不同的聲音。

note 程序中的音調 **P**（介於 0 到 50 的整數），節拍 **d**（介於 0 到 255 的整數）。呼叫 **note (2,2)** 就產生了碰撞的聲音。這種特性未必在每一種電子計算機上適用。

螢幕上兩點間畫一直線的常式 **line** 如下：

```

procedure line(v1, v2: vector);
{ Plot line between two points }

begin { line }
  pencolor(NONE);
  moveto(round(v1[X]), round(v1[Y]));
  pencolor(WHITE);
  moveto(round(v2[X]), round(v2[Y]));
end;

```

建議 (Suggestions)

在 **bouncer** 運轉的時候，如你想要修改一些模式的前提，你就必須把交互作用的數學說明的很清楚，若你修過物理，而且也很有心得，則設計起來，就容易得多了。同時，你也許想改變球和牆壁碰撞方式。例如，當球撞到牆壁時，損失了一些能；也許球旋轉，球彼此交互作用有重大的變化、球和牆壁碰撞作用也有所變化，數學的細節狀況，就由你去掌握。

在發展 **boucer** 的過程中，我們一直忽略了各項變數的度量單位，也許你希望能自由選擇規定之。

也許你允許使用人鍵入數目，以說明盒子的大小，你是有良好的辦法，很容易的告訴他，該盒的形狀大小。

其次，你允許在盒子內放一些障礙物，比如說牆壁、門等，這些在遊戲內很容易發展的，亦允許使用搖桿來操縱牆壁，使其撞及目標。

也許你想發展一般性的畫圖常式，可以在螢幕上表示出任意大小的盒子、障礙物。

加一個命令，允許使用人暫時把螢幕凍結，也可使用列表機畫圖，也可存到磁碟。

最後，你也可設計類似三度空間的問題，模式就更為複雜了，複雜程度端賴所需繪圖的常式的數學了。

isaac 模擬

第二個模擬 **isaac**，在這個例子中，應用到牛頓發明的萬有引力定律。

isaac 的結構與 **bouncer** 很像：每一次執行皆包含若干個運轉，每次運轉皆需要使用人說明一些參數。**isaac** 除了要註明事物的個數、時間間隔、初速度、最初位速之外，尚要鍵入每個事物的質量，圖 7-2，7-3，7-4 為 **isaac** 快速拍照下的運轉狀況。

isaac 主要常式如下：

```
{Ss+}
program isaac;

uses
  applestuff,           { for keypress }
  turtlegraphics,       { for graphics routines }
  transcend,           { for square root }
  {$u apple2:toolstuff.code }
  crtstuff;

const
  MAXOBJECTS = 10;      { Maximum number of objects per run }
  PICSIZE = 5;         { Size of picture array in dots }

type
  coordinate = (X, Y);
  vector = array [coordinate] of real;
  objectrec = record
    mass: real;
    pos, vel: vector
  end;
  picture = packed array [1..PICSIZE, 1..PICSIZE] of boolean;

var
  object: array [1..MAXOBJECTS] of objectrec;
  i, margin, nobjects: integer;
  prompt: array [1..9] of string;
  dt: real;
  showpath: boolean;
  ballpic: picture;

{-----}
{ Modules to be inserted here: }
{   initisaac                     }
{   getiparams                   }
{   runisaac                     }
{-----}
```



```
begin ( isaac )  
  crt(CLEAR);  
  disptitle('isaac');  
  initisaac;  
  for i := 1 to 9 do  
    posstr(prompt[i], margin, 2 + 2 * i);  
  repeat  
    getiparams;  
    if nobjects > 0 then  
      runisaac  
  until nobjects <= 0;  
  crt(CLEAR)  
end.
```

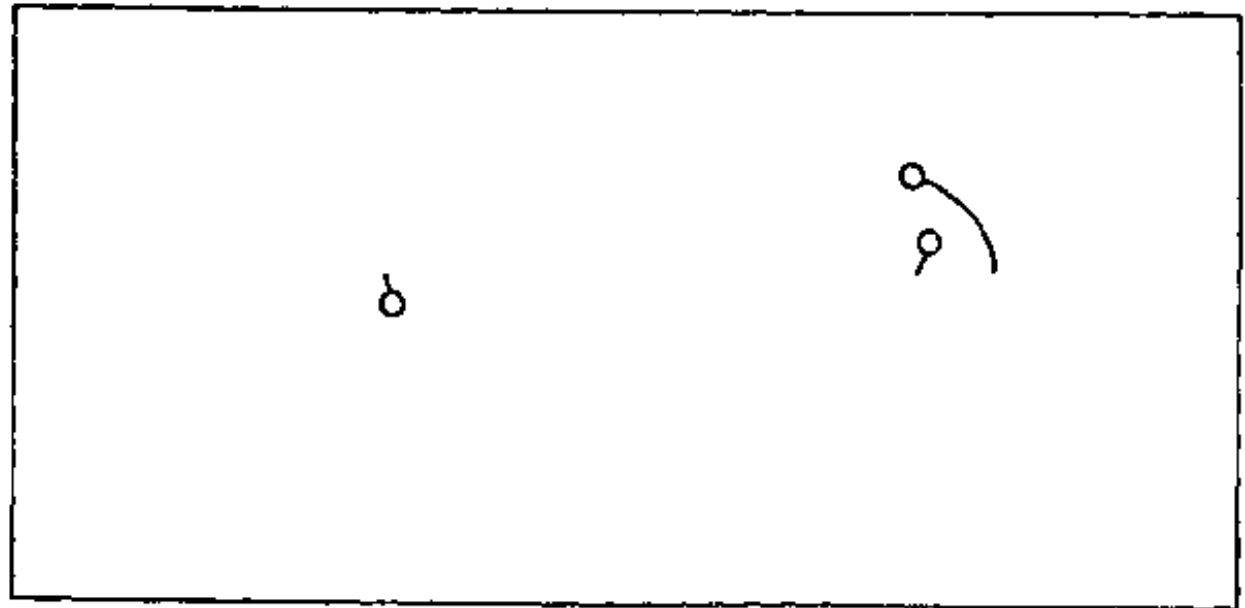


圖 7-2 典型 isacc 重複 12 次後的運轉

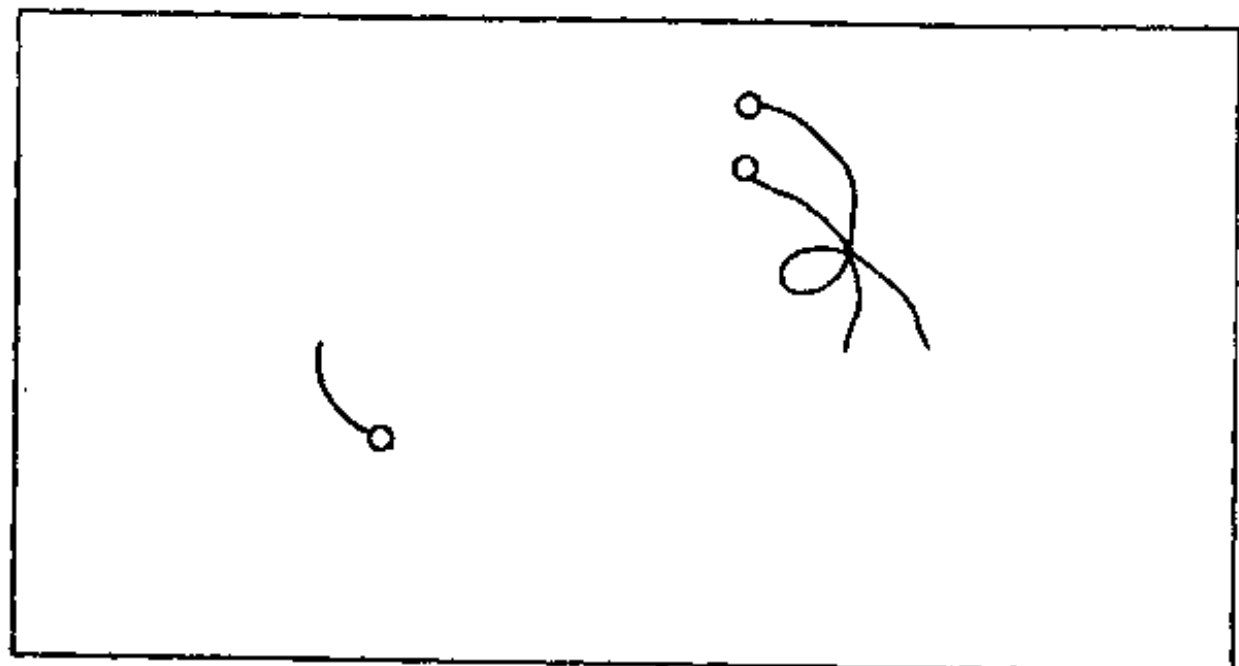


圖 7-3 isacc 重複 64 次後的運轉

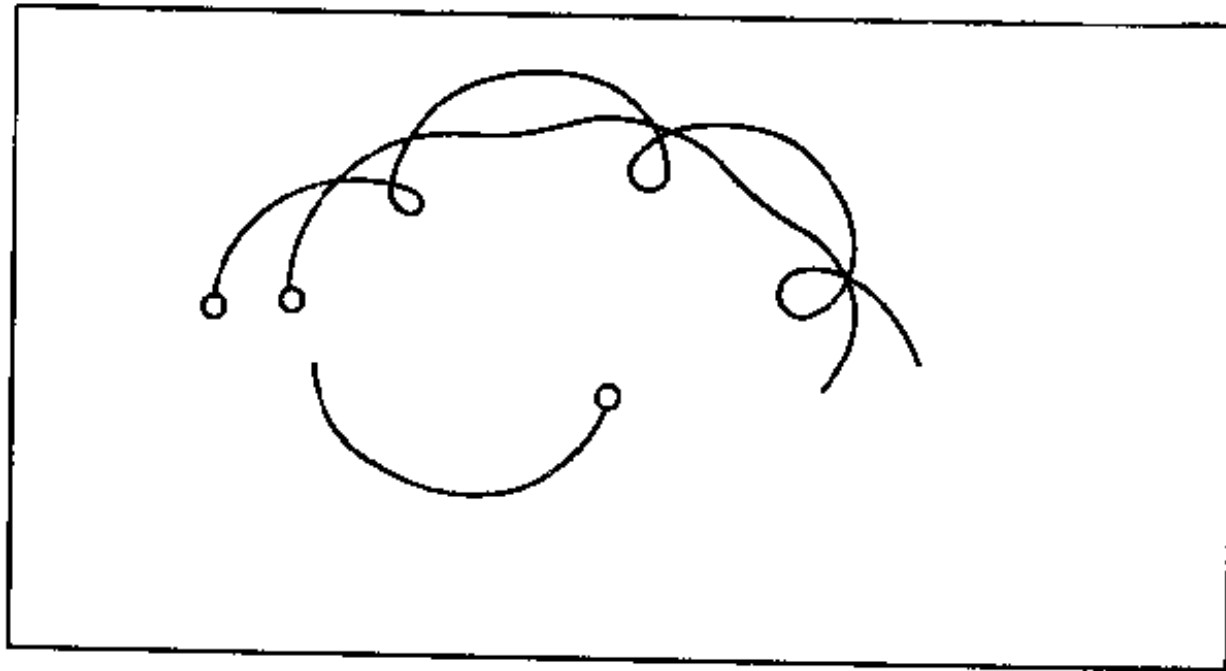


圖 7-4 isaac 重複 175 次後的運轉

注意：在 **uses** 敘述內，附加了一個 **transcend** 單元，**APPLE PASCAL** 把 **transcendental** 函數、置於這個單元內，而不置於編譯器內（成為內建函數）。**isaac** 使用這個單元的平方根函數。

注意，**objectrec** 型態的宣告，就如同 **bouncer** 中 **ballrec** 型態的宣告一樣。差別僅在於 **objectrec** 中包括事物的質量。

initisaac 程序也和 **bouncer** 相類似；它定 **prompt** 陣列、**margin** 度數、**ballpic** 陣列的初值。

initisaac 程序如下：

```
procedure initisaac;  
( Initialize isaac's globals )  
  
var  
    i, j: integer;  
  
begin ( initisaac )  
    prompt[1] := 'Number of objects (0 to exit):';  
    prompt[2] := 'Time interval:';  
    prompt[3] := 'Show object paths(Y/N):';  
    prompt[4] := '    Object No. ';
```

• 7 模擬與漫畫 •

```
prompt[5] := 'Mass: ';
prompt[6] := 'Initial x-position: ';
prompt[7] := 'Initial y-position: ';
prompt[8] := 'Initial x-velocity: ';
prompt[9] := 'Initial y-velocity: ';
margin := (MAXCRTCOL - 29) div 2;
for i := 1 to 5 do
  for j := 1 to 5 do
    ballpic[i, j] := ((i in [1, 5]) and (j in [2..4])) or
                      ((j in [1, 5]) and (i in [2..4]))
  end;
end;
```

getiparams 程序由使用者處取得運轉參數。

```
procedure getiparams;
{ Accept parameters for isaac run }

const
  VMAX = 10.0;      { Maximum abs. val. of initial velocity component }
  XMIN = 0.0;      { Minimum initial x-position }
  XMAX = 279.0;    { Maximum initial x-position }
  YMIN = 0.0;      { Minimum initial y-position }
  YMAX = 191.0;    { Maximum initial y-position }

type
  format = (FIXEDPOINT, SCIENTIFIC);

var
  i: integer;
  s: string;

{-----}
{ Modules to be inserted here: }
{      itos      }
{      stoi      }
{      getint     }
{      getreal    }
{-----}

begin { getiparams }
  nobjects := getint(margin + length(prompt[1]), 4, 0, MAXOBJECTS, 0, FALSE);
  if nobjects > 0 then begin
    dt := getreal(margin + length(prompt[2]), 6, 0.0001, 1.0, 0.0,
                  FIXEDPOINT, 6, 4, FALSE);
    showpath := getboolean(margin + length(prompt[3]), 8, FALSE, FALSE);
    for i := 1 to nobjects do begin
      itos(i, 0, s);
      posstr(s, margin + length(prompt[4]), 10);
    end;
  end;
```

• 高等 Pascal 程式設計技巧 •

```
with object[i] do begin
  mass := getreal(margin + length(prompt[5]), 12, 0.0, 1000.0, 0.0,
    FIXEDPOINT, 6, 1, FALSE);
  pos[X] := getreal(margin + length(prompt[6]), 14, XMIN, XMAX, 0.0,
    FIXEDPOINT, 5, 1, FALSE);
  pos[Y] := getreal(margin + length(prompt[7]), 16, YMIN, YMAX, 0.0,
    FIXEDPOINT, 5, 1, FALSE);
  vel[X] := getreal(margin + length(prompt[8]), 18, -VMAX, VMAX, 0.0,
    FIXEDPOINT, 6, 2, FALSE);
  vel[Y] := getreal(margin + length(prompt[9]), 20, -VMAX, VMAX, 0.0,
    FIXEDPOINT, 6, 2, FALSE);
end
end
end
end;
```

runisaac 運轉與 runbouncer 很類似

```
procedure runisaac;
{ Run isaac simulation }

var
  ch: char;

{-----}
{ Modules to be inserted here: }
{   initgraphics   }
{   plotball       }
{   initirun       }
{   moveobjs       }
{   termgraphics   }
{-----}

begin { runisaac }
  initgraphics;
  initirun;
  repeat
    moveobjs
  until keypress;
  ch := getkey(ch, [chr(0)..chr(127)], FALSE);
  termgraphics
end;
```

重力作用 (How Gravity Works)

牛頓發現一個事物向另一事物 (質量 m) , 距離為 r 落下 , 則其向下的加速度和引力 , 很容易用公式表示出來。

PASCAL 可寫成如下：

$acc := G * m / sqr(r);$

此處 G 是一常數，其與單位有關，用來表示 acc 、 m 和 r 的量，在本章中，令 $G = 1$

根據上面公式的加速度向量，加速度向量的方向，可以求得加速度向量的其他分量。

令事物 1 的位置 (x_1, y_1, z_1) ，吸引另一事物 2 的位置 (x_2, y_2, z_3) ，第一步驟求兩者距離 r ，首先求其向量差如下：

$$(x_1, y_1, z_1) - (x_2, y_2, z_2) = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$$

這新向量可視為事物 1，與事物 2 的相對位置：就像座標系統的原點對於事物 2，則距離為：

$$r := \text{sqr}(\text{sqr}(x_1 - x_2) + \text{sqr}(y_1 - y_2) + \text{sqr}(z_1 - z_2))$$

這是個一般性的公式，可求出坐標系統中任兩點的距離。

其次的步驟是求各個加速度的分量，如 $x - acc$ ， $y - acc$ ， $z - acc$ ，其方向自事物 1 向事物 2，加速度的各分量和其向量長度 acc ，必定與各分量的差有相同的比例，其向量長度 r ，數學式表示法如下：

$$x - acc / (x_1 - x_2) = y - acc / (y_1 - y_2) = z - acc / (z_1 - z_2) = acc / r$$

而 x 一分量如下：

$$x - acc := acc * (x_1 - x_2) / r$$

同理 $Y - acc$ 和 $Z - acc$ 如下：

```
y-acc := acc * (y1 - y2)/r  
z-acc := acc * (z1 - z2)/r
```

自事物 2 吸引事物 1，其事物 1 的加速度計算方式和上述一樣，只要把事物 2 的質量 m ，替換成事物 1 質量 m ，而把 X_1 ， Y_1 ， Z_1 與 X_2 ， Y_2 ， Z_2 互換。

雖然我們討論 3 度（維）空間，我們可做做 **bouncer**，忽略 Z 一分量。

計算的方法 (Calculation Methods)

我們仍然使用等式：

```
p(t + dt) := p(t) + dt * v(t + dt/2)
```

去計算經 dt 時間之後的新位置。

上式中加速度值改變了，不復是常數，而 $v(t + dt/2)$ 這因素求法如下：

```
v(t + dt/2) := v(t - dt/2)  
               + dt * <average acceleration between t - dt/2  
                      and t + dt/2>
```

假定事物的加速度改變的非常慢、圓滑，可估計大約在 $t - dt/2$ 和 $t + dt/2$ 之間，於是公式可改成：

```
v(t + dt/2) := v(t - dt/2) + dt * a(t)
```

這公式之中，在一時間間隔內，事物的加速度必須在我們的估計範圍之內，因此，每運轉一次，初時間定 $t = 0$ ，估計 $dt/2$ 時間的速度如下：

$$v(-dt/2) := v(0) - (dt/2) * a(0)$$

總之，更新事物的位置其方法如下：

```
begin
  calculate relative position vectors
    (using position vectors at time t, p(t))
  for each object
    calculate acceleration vector at time t, a(t)
      (using relative position vectors calculated above)
    calculate "average" velocity vector, v(t + dt/2)
      (using old velocity, v(t - dt/2) and a(t),
      calculated above)
    calculate new position vector at time (t+dt), p(t+dt)
      (using old position, p(t), and average velocity
      calculated above)
  update graphics screen
end-for
end
```

我們必須定初值如下，以確保無誤。

```
begin
  calculate relative position vectors
    (using initial position vectors at time 0, p(0))
  for each object
    calculate acceleration vector at time 0, a(0)
    use it to estimate velocity vector at time (-dt/2),
      v(-dt/2)
  end-for
end
```

更多的資料結構 (More Data Structure)

在計算方面，有一定的次序，第一先求在時間 t 各相對位置向量，再據之求 $t + dt$ 時位置向量，每次計算時，不求加速度，事物的位置也不更新，則加速度就如爬行一般。若無法求

出所有事物的相關位置，就無法更新位置。因此，我們必須有一個資料結構，才能掌握這些所有的相對位置。

最自然的方法，即使用 PASCAL 的矩陣概念，宣告：

```
<type>
  relpos: array [1..MAXOBJECTS, 1..MAXOBJECTS] of vector;
```

於是乎，**relpos** 矩陣掌握住相對位置向量。例如，向量 **relpos[2,4]** 即為事物 2 和事物 4 的位置向量之差，它的兩分量，分別是 **relpos[2,4][x]**、**relpos[2,4][Y]**；我們可以寫成複雜的形式，如 **relpos[2,4,x]**、**relpos[2,4,Y]**。

記住，利用距離 **r** 和 **sgrcr** 計算加速度，只要我們累積計算相對位置向量，在同一時間內，我們還可利用它們去計算這些數量（相對位置向量）。我們依然靠矩陣掌握之：

```
<type>
  r, rsq: array [1..MAXOBJECTS, 1..MAXOBJECTS] of real;
```

數字 **r[3,1]** 表事物 3 與事物 1 間的距離，事物 5 和事物 6 的距離平方根，就以 **sqr[5,6]** 表示之。

下列各對稱關係，足以掌握各陣列的每一元素：

```
relpos[j, i, k] = -relpos[i, j, k]
r[j, i] = r[i, j]
rsq[j, i] = rsq[i, j]
```

relpos 是由兩事物的抽象坐標求出的，對調 **i** 和 **j**，則正負符號亦隨之更動如下：

```
relpos[j, i, k] := (object[j].pos[k] - object[i].pos[k]);
                 := - (object[i].pos[k] - object[j].pos[k]);
                 := - relpos[i, j, k]
```


initrun程序

現在，我們準備設計 `initrun`。只要呼叫 `Plotball` 常式，就可以繪得每一事物最初位置。正如前面的描述一樣，我們必須估計，時間 $-dt/2$ 時各物體的速度，以 PASCAL 撰寫程式則如下：

```

procedure initrun;
( Initialize isaac run, do initial velocity estimate )

var
  relpos: array [1..MAXOBJECTS, 1..MAXOBJECTS] of vector;
  r, rsq: array [1..MAXOBJECTS, 1..MAXOBJECTS] of real;
  i, j: integer;
  k: coordinate;
  acc: vector;
  temp: real;

begin ( initrun )
  for i := 1 to nobjects - 1 do
    for j := i + 1 to nobjects do begin
      for k := X to Y do begin
        relpos[i, j, k] := object[j].pos[k] - object[i].pos[k];
        relpos[j, i, k] := -relpos[i, j, k]
      end;
      rsq[i, j] := sqr(relpos[i, j, X]) + sqr(relpos[i, j, Y]);
      r[i, j] := sqrt(rsq[i, j]);
      rsq[j, i] := rsq[i, j];
      r[j, i] := r[i, j]
    end;
    for i := 1 to nobjects do begin
      acc[X] := 0.0;
      acc[Y] := 0.0;
      for j := 1 to nobjects do
        if i <> j then begin
          temp := object[j].mass/rsq[i, j]/r[i, j];
          acc[X] := acc[X] + temp * relpos[i, j, X];
          acc[Y] := acc[Y] + temp * relpos[i, j, Y]
        end;
      with object[i] do begin
        vel[X] := vel[X] - acc[X] * dt/2.0;
        vel[Y] := vel[Y] - acc[Y] * dt/2.0;
        plotball(pos)
      end
    end
  end
end;

```

注意，在計算 `relpos`、`r` 和 `sqr` 矩陣時，`initrun` 充份利用矩陣的對稱性，只計算一半的元素就夠了。

moveobjs 搬移物體程序

根據舊的數值求出的新位置和速度，並把物體的這些變化搬移到螢幕上以圖顯示，需要設計moveobjs 如下：

```
procedure moveobjs;
{ Calculate new positions and velocities, move balls on screen }

var
  relpos: array [1..MAXOBJECTS, 1..MAXOBJECTS] of vector;
  r, rsq: array [1..MAXOBJECTS, 1..MAXOBJECTS] of real;
  i, j: integer;
  k: coordinate;
  acc, newpos: vector;
  temp: real;

{-----}
{ Modules to be inserted here: }
{      line      }
{-----}

begin { moveobjs }
  for i := 1 to nobjects - 1 do
    for j := i + 1 to nobjects do begin
      for k := X to Y do begin
        relpos[i, j, k] := object[j].pos[k] - object[i].pos[k];
        relpos[j, i, k] := -relpos[i, j, k];
      end;
      rsq[i, j] := sqr(relpos[i, j, X]) + sqr(relpos[i, j, Y]);
      r[i, j] := sqrt(rsq[i, j]);
      rsq[j, i] := rsq[i, j];
      r[j, i] := r[i, j];
    end;
    for i := 1 to nobjects do begin
      acc[X] := 0.0;
      acc[Y] := 0.0;
      for j := 1 to nobjects do
        if i <> j then begin
          temp := object[j].mass/rsq[i, j]/r[i, j];
          acc[X] := acc[X] + temp * relpos[i, j, X];
          acc[Y] := acc[Y] + temp * relpos[i, j, Y];
        end;
      with object[i] do begin
        for k := X to Y do begin
          vel[k] := vel[k] + acc[k] * dt;
          newpos[k] := pos[k] + vel[k] * dt;
        end;
        plotball(pos);
        plotball(newpos);
        if showpath then
          line(pos, newpos);
        pos := newpos;
      end;
    end;
  end;
end;
```

在 isaac 的運轉中，選擇運轉參數，使其在螢幕顯示有趣畫面，並不容意。表 7-1 列舉一些值得注意的輸入參數值。

表 7-1 issacc 的運轉參數

Run No.	Object No.	Mass	Position		Velocity	
			X	Y	X	Y
1	1	1000.0	140.0	95.0	0.0	-0.002
	2	1.0	275.0	95.0	0.0	2.0
2	1	1000.0	5.0	95.0	0.0	0.5
	2	500.0	275.0	95.0	0.0	-1.0
3	1	1000.0	80.0	95.0	0.0	-1.0
	2	400.0	209.0	95.0	0.0	1.0
	3	100.0	229.0	95.0	0.0	6.0
4	1	1000.0	5.0	95.0	0.0	1.0
	2	1000.0	275.0	95.0	0.0	-1.0
	3	1.0	140.0	95.0	0.0	0.0
5	1	1000.0	140.0	95.0	0.0	-0.03
	2	10.0	230.0	95.0	0.0	3.33
	3	0.1	185.0	173.0	-2.89	1.67

運轉 1：輕物體圍繞一較重物體的軌道轉，此類似地球圍繞太陽運動。

運轉 2：質量相當的兩物體互相在軌道運動；真實世界的雙子星系統就為等量物體，在彼此的軌道運動。

運轉 3：三個物體的模擬：一個輕物體圍著一個重物體的軌道，而這兩物體又依次繞另一更重的物體之軌道（見圖 7-2，7-3，和 7-4）。太陽、地球、和月亮系統和本例相似。

運轉 4：兩個重質在彼此軌道運行，它們中間有一個輕物體，在感覺上它們並無加速度，但是在 isaac 的不精確的計算之下，導致該輕物體有些微的偏離中心，因此，它以很均勻的速度移向其重物體之一，最後完全脫離它最初的位置。

運轉 5：一輕物體（例如小遊星），在一較重物體（行星）的近乎圖形的軌道之前，圍著一重得多的物體（恆星）運轉。這三者形幾乎成一個等邊三角形，小遊星、行星、和軌道形

成大約 60° 的角，看起來似乎是行星吸引小遊星或是在另一軌道上要捕捉小遊星，或是把小遊星自另一軌道擲開，然而，這種情況是不會發生的，小遊星總是在行星的前面，這輕物體可說是在穩定拉格朗日 (Lagrangian) 點 L4，另一穩定拉格朗日點 L5，而 L5 在行星之後，其在軌道上形成近乎 60° 。

木星 (Jupiter) 系統類似運轉 5，有一群小遊星在對應 L4 和 L5 的穩定軌道上，或在木星之前，或在之後近乎 60° 。

讀者也許在表 7-1 看到一件共同事情：向量動量的總和差不多是 0，如沒有這個限制，則該系統終究要搬離螢幕，此亦即在表 7-1 中並不將 x 一速度是初值。很容易藉定把所有物體速度的一個分量為 0，就能設定一個動量為 0 的系統。

更多的建議 (More suggestions)

幾乎所有對於 **bouncer** 實驗的建議，都能應用於 **isaac**，此外，尚允許因時而改變物體的質量。

isaac 有一個問題，即一個物體偶爾在螢幕上消失，促使使用人遺忘了它（並不會使 **APPLE PASCAL** 在執行時發生錯誤。）你可設計一個方法，讓所有的物體留在螢幕上，若有一個離開螢幕，則自動的重新製圖，若所有物體都只占據螢幕的一小塊，你亦可自動製圖。

當兩物體通過另一物體時靠得太近，**isaac** 發生其他問題，則在 **dt** 時間內，被計算的加速度變化太大，這與我們的大前提——在時間間隔內，**isaac** 的改變相當緩慢、平滑。結果，這個模擬失準。自動偵測這個狀況，並減短 **dt**，可以改進之。（反之，若無碍這模擬的準確性，也可增長 **dt**）。

模擬的速度受到計算的限制，可以簡化，先計算，再由圖案常式顯示其結果。

你也許希望以不同的定律實驗去求加速度，則一定要弄清楚法則，也就是說兩物體間有不同的相關距離。也可在它們中間增加其他的引力、如電磁場。在真實的世界裡，許多重物體都有自己的磁場，在與地心引力比較起來，這些磁場往往可以忽略掉。假如，你在電磁場裡，去計算電的粒子運動狀況，則重力效用却可忽略不計。

推薦閱讀 (Recommended Reading)

- 1 N-Roberts , D-Andersen, R.Deal 和W.Shaffer 合著電子計算機模擬介紹 (Introduction to Computer Simulation) 。
- 2 W.R.Bennett, Jr. 著 Scientific and Engineering problem-Solving with the Computer 。
- 3 R-Feynman 著 The Feynman Lectures on physics 第一冊第九章。
- 4 R.Myers 著 Microcomputer Graphics 。

8

電子工作表格

(*The Plane Truth: An Electronic Worksheet*)

我們設計一個類似 Visi Calc, Super Calc, Calcstar, Multiplan 的「電子的工作表」程式 **Pascale**。

所謂工作表 (worksheet) 程式有下列共同特性。它是格 (cells) 的長方形陣，由行和列組成，CRT 螢幕就像一個窗子顯示一部份的工作紙 (通常工作表很大，無法一次顯示完畢)。簡單的命令可以在任何方向，捲動窗子，看該紙的任意部份。程式的使用者，藉移動游標 (cursor) 到所希望的格子，寫到工作表並鍵入資訊。使用者在格內可寫一個標記 (label) —— 任意字元串，一個數、或一個公式。

一個格子公式 (cell formula) 與其他的格子有關，所以在程式裡、其回答「what-if」問題時尤其有用，比方說，工作表可以根據利息計算一大筆貸款分期攤還的結果，只需改變利率，便能改變其他項目。

工作表程式允許電子計算機去解決銀行存簿、預算和圖形表示稅務等問題。它可以取代用紙筆去計算。

Pascale 與一般商界用的程式不同，它速度慢，對記憶體也無法有效的運用，也無法提供太多的功能，從另一方面來說，**Pascale** 具有可塑性，它可以修改，依據讀者的需要，自行

增、刪、或改變某些特性。

Pascalc作業方式 (How Pascalc Works)

下述簡單介紹Pascalc的作業方式，細節有待設計該程式時，再作進一步描述。假定我的螢幕是 80×24 ，當然它可根據需要允以調整。

開始的時候，螢幕就像圖8-1所示。CRT顯示工作表前面7行和第8行的一部份，這行些數在螢幕的上方都有標記。（注意，工作表的行位與螢幕的行位不同。）一個工作表的行位，往往占據數個螢幕的行位。列位的標記，則在螢幕左方，它顯示著最前面的列，游標占著行1、列1、列1的格子上（左上角），由【1，1】標明之。

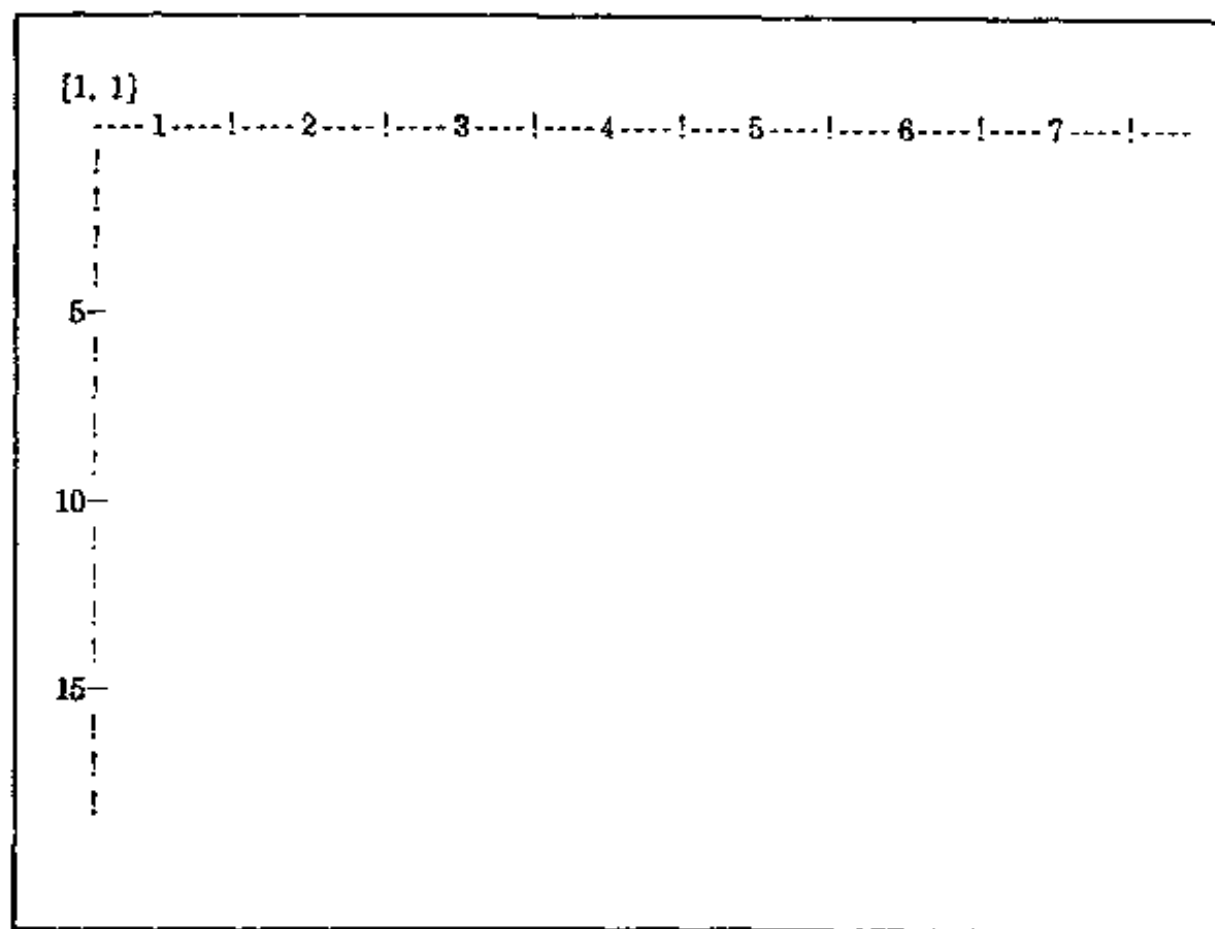


圖 8 - 1 Pascalc 螢幕最初顯示

由最初位置，你可不拘次序的做下列四件事情：

移動游標 (Move The Cursor)

藉鍵盤移動游標，但不可鍵入命令，要求它離開工作表 (sheet)，否則就錯誤，並發「嗶」的聲音。

鍵入標記 (Enter a Label)

在移動標記之後，可在工作表上註記之。你可鍵入字母或雙括號字元 < , , >。鍵入字母，表示該標記的第一字母，就是鍵入的字母，而 < , , > 表示，你鍵入的標記，第一個字元不是字母。

鍵入標記經常他用到字串，< BACKSPACE > 表示擦掉鍵入的最後一字母，< DELETE > 與 < BACKSPACE > 同意。而 < RETURN > 表鍵入工作完畢。標記可使用任何可印得出的字元，標記鍵完後，游標即占據新的格子，即使標記已超過現行數的寬度，它依然把標記全部顯示之。(嚴格的說，若標記的前面或後面的字元，超出窗外，則只有一部份的標記在CRT上顯示。

鍵入一個數或公式 (Enter a Number or Formula)

你可鍵入一個數、或一個公式，以代替標記。事實上，**pascal** 並不能區別出數字和公式。若你想鍵入一個字元，來代表你想鍵入一個公式，只需鍵入一個自 < 0 > 到 < 9 > 的數位、< + > 或 < - > 符號，十進位小數點，左括號 < (>、或左中括號 < [>。

公式鍵入方式與第四章的 **calc** 很像，最大差別，在於識別號 (identifiers)，在 **calc** 中的識別號是文數字串 (alphanumeric strings) 例如，**APPLE**，**SIDEL**，等等。它允許用式子先算好的結果。在 **Pascal** 中的格子，則使用「先算好的結果」取代識別號。

格子的參考型態分爲兩種——絕對和相對。絕對型態以

【< column > , < row >】形式表示，行數或列數記在中括號內，並由逗號分開。例如，【4 , 9】表示在Pascal 內使用行4、列9代表格位。而相對型態，則在行數、列數前面冠以正、負號。例如，在【19 , 12】格子的公式中包含參考【-3, +5】，的意思是在【19 , 12】格子的左方3行，下面5列，換言之，現行位置在【16 , 17】。

使用相對格子參考是非常簡單的，可藉移動浮標指示該格位，詳於getformstr和getrec 常式再討論。

SUM 函數、用來把任意個元素相加，例如，式子SUM (【1 , 1】 : 【1 , 10】)代表【1 , 1】+【1 , 2】+【1 , 3】+……+【1 , 10】，詳於后討論。

鍵入指令 (Enter a Command)

Pascal 提供的命令有：改變工作表的格式、擦掉各格子、自工作單某一點抄到另一點、等等。Pascal 簡表見表8-1，從表面上看來，Pascal 並不是很複雜的程式。

表 8-1 Pascal 命 令

命 令	鍵 序	描 述
Recalculate	!	重新計算工作單。
Go To	7	移動浮標到特定的格位。
Copy	/C	自工作單的一部份抄到另一部份。
Erase	/E	擦掉一格，一或多列、一或多行，或整張單子。
Format	/F	顯示選擇參數。

Insert	/I	在工作單的行或列上，插入一個或多個空白。
Load	/L	自磁碟中把上次存入的工作單載入記憶體。
Memory	/M	顯示未使用的記憶體。
Print	/P	在列表機列印sheet。
Quit	/Q	離開程式。
Save	/S	把sheet存到磁碟。
Toggle	/T	撥鈕重計參數。

資料結構——稀疏矩陣 (The Sparse Matrix)

Pascal 工作表由 63 行組成，每行有 256 列格子，可由二維陣列表示如下：

```
<var>
cell: array [1..MAXCOLS, 1..MAXROWS] of cellrec;
```

而 **cellrec** 資料型態是一個記錄 (record)，陣列中 **MAXCOLS** × **MAXROWS** 元素總數超過 16,000，每一個 **cellrec** 的記錄約 10 個位元組，該矩陣需要至少 16,000 個位元組的記憶體，這超過很多電腦的容量。

我們當然可以減少 **MAXCOLS** 和 **MAXROWS**，直到 **cell** 陣列在記憶體內能有效的掌握。此外，就需一個資料結構——稀疏矩陣來解決之。

所謂一個稀疏矩陣，是一個「大部分元素為空」的矩陣，

既然有許多元素是空的 (empty)、就沒有理由浪費記憶去儲存。因此，可以重新安排成：每個行 (column) 都有一個行的頭 (header)，有一個指標指著最體的非零元素，假如，該行裡沒有非空元素，則指標為 NIL，它並不指向任何元素。每一行的每一元素，都有一個指標指向下一非空元素，假如某一元素是該行最後一個非空的元素，則其指標為 NIL。

對列而言，也有同樣的結構，任一陣列都有一列的頭，指標指向該列的最左的格子，若該列沒有格子了、就指向 NIL，每一元素有一個「右指標」指向該列右邊一個元素。

圖 8-2 顯示一個假定的 5×5 矩陣資料結構，其中 $[1, 1]$ ， $[1, 2]$ ， $[1, 5]$ ， $[3, 2]$ ， $[3, 5]$ ， $[4, 1]$ ， $[5, 2]$ 皆為非空 cells，行的頭指標為各盒子頂上的數字，而列的頭指標，例為圖形左方的數字。電覽符號代表指標為 NIL，每一 cell 都有兩個指標指著它，一個是它下方行上，另一是列的右方。

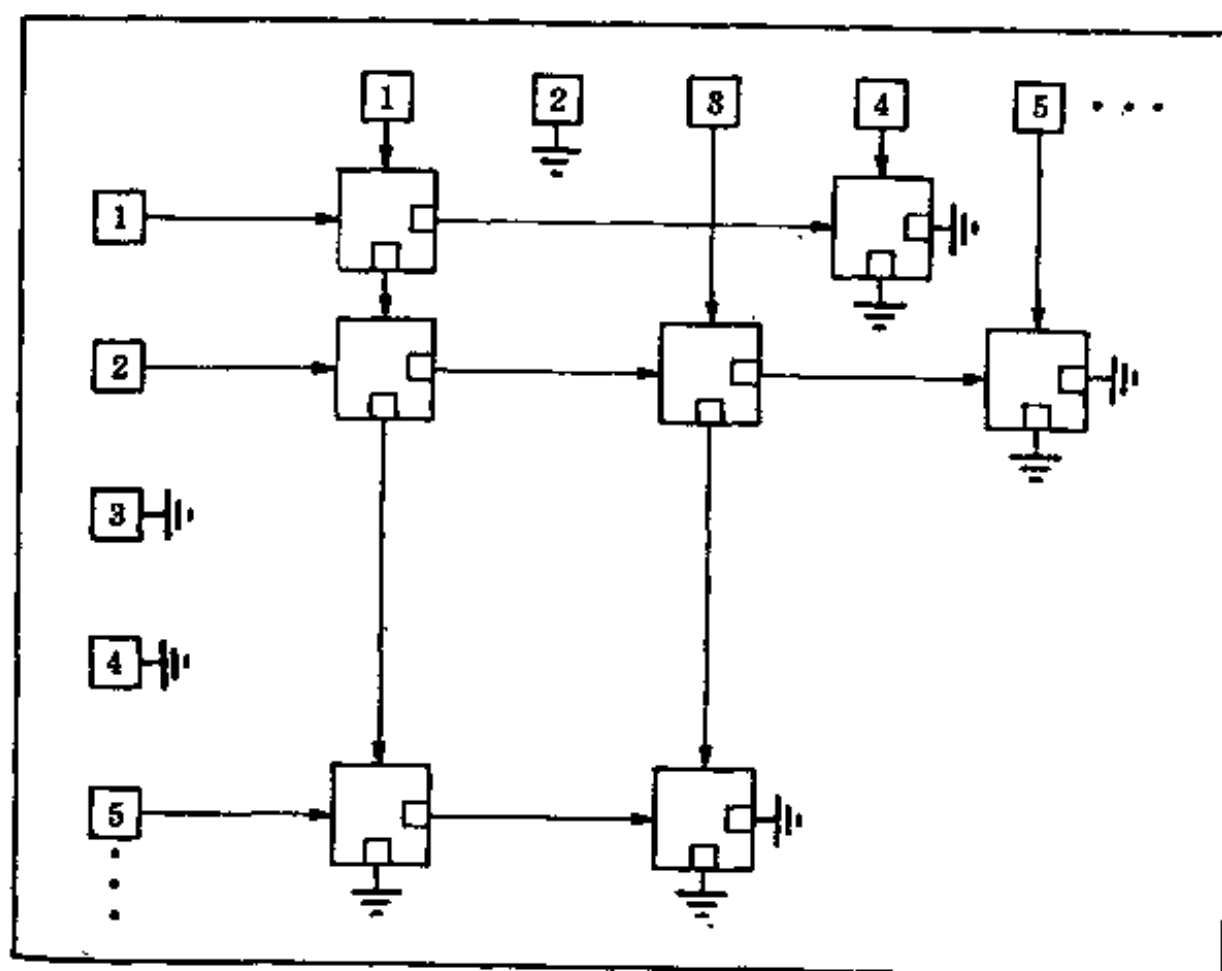


圖 8-2 稀疏矩陣資料結構

在PASCAL 程式，可建立資料結構如下，首先定義 **cellptr** 和 **cellrec** 型態：

```
<type>
  cellptr = ^cellrec;
  cellrec = packed record
    cellcol: 1..MAXCOLS;
    cellrow: 1..MAXROWS;

    { other stuff }

    rightptr, downptr: cellptr
  end;
```

除了指向下方和右方的 **cells** 之外，每一個 **cell** 記錄，都有它們自己的列和行坐標。**{other stuff}** 說明我們沒有定義的 **cell** 欄位。

最後定義行的頭與列的頭如下：

```
<var>
  colptr: array [1..MAXCOLS] of cellptr;
  rowptr: array [1..MAXROWS] of cellptr;
```

有了上述的資料結構，再設計一個常式 **findcell**，去找 **cell**：

這常式藉著行 **c**、列 **r**，回復 **cell** 一個指標，若在工作表上並沒有定義 **cell**，則 **findcell** 就回轉一個 **NIL** 指標，**findcell** 是利用 **cellcol** 值、和列 **r** 來找 **cell**；首先，**cell** 被 **rowptr[r]** 所指，在列上向右搬移，直到找到 **cell**，或確定找不到 **cell** 為止。

若使用稀疏矩陣只是省記憶空間，而又與「大部分元素為空的」大前提相違的話，就得仰賴一個簡單二維空間的陣列結構了，它比稀疏更節省記憶空間。

```
function findcell(c, r: integer): cellptr;  
( Find cell on sheet, return pointer to it or NIL if not there )  
  
var  
  cp: cellptr;  
  done: boolean;  
  
begin ( findcell )  
  if (c < 1) or (c > MAXCOLS) or (r < 1) or (r > MAXROWS) then ( off sheet )  
    findcell := NIL  
  else begin  
    cp := rowptr[r];  
    done := FALSE;  
    while (cp <> NIL) and not done do  
      if cp^.cellcol < c then  
        cp := cp^.rightptr  
      else begin  
        done := TRUE;  
        if cp^.cellcol <> c then  
          cp := NIL  
        end;  
        findcell := cp  
      end  
    end;  
  end;  
end;
```

Cell記錄

前面概略的定義了包含 { **other stuff** } 的資料型態 **cellrec**，下一步驟即描述該記錄。

由上述討論已知一個非空 **cell** 包括一個標記或一個公式，無論是那一種狀況，一個 **cell** 必須有一個欄位，它描述螢幕顯示狀況。當該 **cell** 包含一個標記 (**label**)，則顯示欄就是標記的本身，若該 **cell** 包含一個公式，則該欄就包括一個真正被格式化的字串，它也是該公式的數值。

上述方法可寫如下頁：

此處，增加一欄 **display** 它包括一個字串，在 **cell** 的位置顯示。這不是我們使用的方法，因為太浪費記憶了。在程式中宣告字串，就需 81 個位元組，我們缺設 (**default**) 該字

```
<type>
  cellrec = packed record
    cellcol: 1..MAXCOLS;
    cellrow: 1..MAXROWS;

    { other stuff }

    display: string;

    rightptr, downptr: cellptr
  end;
```

串 80 字元，另一字元是爲了長度位元組。通常 **display** 欄遠小於 80 字元，平常字長 10 字元，於是每產生一個新的 **cell** 就浪費 70 個字元。

假使減少 **display** 欄，固然可以節省記憶空間，却無法顯示 80 字元了。

在 PASCAL 程式裡，如果定義一個固定記憶體的變數，在程式的執行過程中，都不能改變，在這種情形下，就必須定義其爲最大長度。

以上的問題的解決方案有二：第一，不要定義資料型態而使用指標：

```
<type>
  cellrec = packed record
    cellcol: 1..MAXCOLS;
    cellrow: 1..MAXROWS;

    { other stuff }

    display: stringptr;

    rightptr, downptr: cellptr
  end;
```

此處定義 **stringptr** 爲一字串的指標。

```
<type>
  stringptr = ^string;
```

這樣並未解決問題，却幾近美好了，本章後面將提出第二種方案。

格式 (Forma (Formatting))

在設計用來儲存公式 cells 的資料結構前，必須考慮，如何把公式 cells 的數值翻譯成顯示字串。

第一，考慮如何格式化一數值。在前一章裡顯示小數點後面的數位它是多麼的有用。第四章裡它對 **calc** 程式也非常有用，同時我們也希望有一欄寬度參數，來說明顯示數字時所需要最小字元的個數。

formatrec 如下

```
<type>
  formatrec = packed record
    fmt: format;
    width: MINWIDTH..MAXWIDTH;
    ndigs: MINDIGS..MAXDIGS
  end;
```

在 **calc** 裡 **format** 型態如下：

```
<type>
  format = (FIXEDPOINT, SCIENTIFIC);
```

這三個參數在「格式化」足夠顯示數目了。例如，幣值可以由 **ndigs** 參數 2 和 **fmt** 為定點 (FIXED POINT) 設定元與分的形式，來顯示之。如 **ndigs** 為 0，顯示值就被捨成最

接近的整數。使用 **width** 參數向右調整一個數的行，使其有足夠長度，若是也把行裡安排十進位小數點。如向左調整可藉設定 **width** 參數成 0 實行之。

我們的目的是數字彈性的顯示，我們允許可任意格式化工作表的數，**cell** 的格式化可以獨立執行之。我們設計的目的是容易使用：使用者不必說明 **sheet** 上的每一公式 **cells** 的格式化參數。

我們處理方式如下：使用者可以說明任何公式 **cell** 的格式，如使用者不允以說明，則 **cell** 就以其現行所在的行缺設之 (**default**)。

行的格式處理亦類似，使用者可對一個行註明為固定格式，否則其就以其現行的 **sheet** 格式顯示之。

width 控制 **sheet** 行的寬度，就藉更改行的格式之寬度參數，而達到更改 **sheet** 行的寬度，也可利用缺設 **sheet** 格式來更改所有的 **sheet** 行。

定記憶大小 (Formula Storage)

在 **cellrec** 資料型態中，定義指標 **fp** 是一個包括所有公式資訊的記錄：

```
<type>
  cellrec = packed record
    cellcol: 1..MAXCOLS;
    cellrow: 1..MAXROWS;
    fp: formptr;
    display: stringptr;
    rightptr, downptr: cellptr
  end;
```

而 **formptr** 型態定義如下：

```
<type>
  formptr = ^forminfo;
```

Pascal 很容易分辨出 **cell** 包含一個標記或一個公式，若它是一個標記，則指標 **fp** 是 **NIL**；否則 **fp** 指向一個有效的 **forminfo** 記錄。假如該 **cell** 包含一個標記，這個引數就得到一個附加的益處，那就是不必額外的記憶體來儲存公式的資訊。

forminfo 記錄結構如下：

```
<type>
  forminfo = packed record
    usecolfmt: boolean;
    cellstat: xresult;
    cellfmt: formatrec;
    cellval: xreal;
    formula: stringptr
  end;
```

如果 **cell** 的值根據現行所在的行格式顯示，則 **usecolfmt** 是 **TRUE**；若它是 **FALSE**，則它的格式就由 **cellfmt** 定之。

cellstat 欄是用來指示「在 **cell** 的公式求值的時候是否有錯誤發生？」。**xresult** 型態，我們曾在 **calc** 裡用過，它是用來指出溢出、超下限、或除以 0 等。

cellval 欄包括「被鍵入的公式，計算出來的值」。由於 **xreal** 的範圍和其精度容易控制、也很容易避免溢出、超下限的錯誤，所以選擇 **xreal** 資料型態來代表數的量。此外，在 **calc** 程式中，可以盜用一些常式。

最後，**formula** 欄包含一個指標，它指向使用者鍵入的字串，使用字串指標的理由，與 **cellrec** 記錄中使用 **display** 欄的理由同：節省記憶體空間。

Pascalc 主常式

我們已將 Pascalc 的大部份資料型態定義好了，它的主要常式如下：

```

($s+)
program pascalc;
{ Electronic Spreadsheet }

uses
    {$u apple2:toolstuff.code } crtstuff, fixstuff, intstuff, textstuff;

const
    { Sheet constants }
    MAXROWS = 255;      { Number of rows on sheet }
    MAXCOLS = 63;       { Number of columns on sheet }
    XORG = 4;           { CRT column of first displayed sheet column }
    YORG = 2;           { CRT row of first displayed sheet row }

    { Memory management constants }
    RESERVE = 1950;     { Number of words to not allocate }
    HDRSIZE = 4;        { Size of header in bytes }

    { Constants from calc }
    MINEXP = 0;         { xreal minimum exponent }
    MAXEXP = 127;       { xreal maximum exponent }
    EXCESS = 64;        { xreal exponent excess }
    REGSIZE = 26;       { 2 * FIXSIZE + 2 }
    IDSIZE = 8;         { Maximum identifier length }

    { Formatting constants }
    MINWIDTH = 0;       { Minimum format width }
    MAXWIDTH = 76;      { Maximum format width, MAXCRTCOL - XORG + 1 }
    MINDIGS = 0;        { Minimum format digits }
    MAXDIGS = 11;       { Maximum format digits, FIXSIZE - 1 }

    { Printing constants }
    PCOMSIZE = 15;      { Printer command string size }
    MAXPLEN = 999;      { Maximum page length }
    MTOP = 2;           { # of blank lines between top of page & 1st line }
    MBOT = 2;           { # of blank lines between last line & bottom }

type
    { Types from calc }
    xreal = record
        expo: MINEXP..MAXEXP;
        frac: fixed
    end;
    register = integer[REGSIZE];
    format = (FIXEDPOINT, SCIENTIFIC);
    identifier = string[IDSIZE];
    xresult = (OK, OVERFLOW, UNDERFLOW, ZERODIVIDE);

```

```

( Printing types )
pcommand = string[PCONSIZE];

( Formula information types )
stringptr = ^string;
formatrec = packed record
    fmt: format;
    width: MINWIDTH..MAXWIDTH;
    ndigs: MINDIGS..MAXDIGS;
end;
formptr = ^forminfo;
forminfo = packed record
    usecolfmt: boolean;
    cellstat: xresult;
    cellfmt: formatrec;
    cellval: xreal;
    formula: stringptr;
end;

( Cell information types )
cellptr = ^cellrec;
cellrec = packed record
    cellcol: 1..MAXCOLS;
    cellrow: 1..MAXROWS;
    fp: formptr;
    display: stringptr;
    rightptr, downptr: cellptr;
end;

( Memory management types )
headerptr = ^header;
header = record
    size: integer;
    next: headerptr;
end;

var
( Sheet variables )
autocalc, colcalc: boolean;
sheetfmt: formatrec;
curscol, cursrow, firstrow, lastrow, firstcol, lastcol: integer;
nrows, inpline, statline: integer;
curcp: cellptr;
colptr: array [1..MAXCOLS] of cellptr;
rowptr: array [1..MAXROWS] of cellptr;
dashes: string;

( Column variables )
colpos: array [1..64] of integer;           { 64 = MAXCOLS + 1 }
colfmt: array [1..MAXCOLS] of formatrec;
usesheetfmt: packed array [1..MAXCOLS] of boolean;

( Calculation variables )
zero: xreal;
ten: array [0..REGSIZE] of register;

```

```

{ Command characters & sets }
upkey, downkey, leftkey, rightkey: char;
cursorkeys, labelkeys, formkeys, cmdkeys, legalkeys: charset;
cadchars, formchars: charset;

{ Printing variables }
pagelen, lmarg, rmarg, printx1, printx2, printy1, printy2: integer;
pinit, pterm: pcommand;

{ Memory management variables }
freelist: headerptr;

{ Control-flow variables }
alldone: boolean;
ch: char;

{ Scalar-type to text conversion variables }
fatchar: array [format] of char;
boochar: array [boolean] of char;
statchar: array [xresult] of char;

```

```

{-----}
{ Modules to be inserted here: }
{      initpc      }
{      drawsheet   }
{      labelrows   }
{      labelcols   }
{      setcolpos    }
{      min          }
{      max          }
{      disprow      }
{      disp sheet   }
{      findcell     }
{      setcursor    }
{      movecursor   }
{      memout       }
{      ungetkey      }
{      initalloc    }
{      alloc        }
{      free         }
{      storestr     }
{      erasestr     }
{      storeform    }
{      eraseform    }
{      storecell    }
{      erase cell   }
{      getlabel     }
{      setparam     }
{      stoc         }
{      ctos         }
{      xnorm        }
{      xadd         }
{      calc cell    }
{      recalc       }
{      format cell  }
{      reformat     }
{      getformula   }
{      docommand    }
{-----}

```

```
begin ( pascalc )
  initpc;
  drausheet;
  setcolpos;
  labelcols(1);
  labelrows(1);
  disp sheet(1, 1);
  setcursor(1, 1);
  if initalloc(RESERVE) then
    repeat
      if getkey(ch, legalkeys, FALSE) in cursorkeys then
        movecursor(ch)
      else if ch in labelkeys then
        getlabel(ch)
      else if ch in formkeys then
        getformula(ch)
      else if ch in cmdkeys then
        docommand(ch)
    until alldone
  else
    remark('Not enough memory to run program');
  crt(CLEAR)
end.
```

至於 **uses** 敘述裡引用 **instuff** 單元，該單元我們不曾定義過，在前幾章裡，**instuff** 單元包含了 **itos**、**stoi**、和 **getint** 等常式，如你的 PASCAL 系統，沒有提供這些分開編譯的程序或資料宣告，則你的原始程式就必須包含這些常式和宣告資料，並與程式的其他部份一起編譯。

因 **Pascalc** 是個相當大的程式，它把常數、型態、變數分族式的宣告，並加註說明，表 8-2、8-3、和 8-4 對於大部份的全盤變數 (global variables) 加以簡單描述。表 8-2 描述 **Pascalc** 的 **sheet** 變數。表 8-3 說明控制行的格式的各個行變數。表 8-4 描寫命令字元和集合。

表 8-2 sheet 變 數

名 字	型 態	描 述
autocalc	Boolean	每改變一次公式，sheet 即自動重新計算時，其值 TRUE 。若需要人工操作其值則為 FALSE 。
colcalc	Boolean	由行控制 sheet 的再計算，值是 TRUE ；若由列控制，其值為 FALSE 。
sheetfmt	formatrec	提供 cells 缺設 (default) 格式參數。
curscol	integer	游標現行位置的行、列坐標。
firstrow	integer	在螢幕顯示時第一和最後的 sheet 列。
firstcol	integer	在螢幕顯示 sheet 的第一和最後一行。
nrow	integer	螢幕顯示 sheet 的列數。
inpline	integer	輸入所在的 CRT 行數。
statline	integer	程式狀態在 CRT 顯示時所在的行數。
curcp	cellptr	指向 cell (若 cell 是空時，其為 NIL) 。
colptr rowptr	array of	sheet 行的頭、列的頭。
dashes	string	顯示行的標記。

表 8-3 行 變 數

名 字	型 態	描 述
colpos	array of integer	對於每一行的相對起點位置。
colfmt	array of integer	對於每一行的格式參數。
usesheetfmt	array of Boolean	若行使用缺設 sheet 格式，陣列元素為 TRUE ，否則陣列元素為 FALSE 。

表 8-4 命令字元和命令集合

upkey	char	游標向上移動一個sheet列的鍵。
downkey	char	游標向下移一個 sheet列的鍵。
leftkey	char	游標向左移一個 sheet行的鍵。
rightkey	char	游標向右移一個 sheet行的鍵。
cursorkeys	charset	「游標 4 個移動的鍵」集合。
labelkeys	charset	「合法的標記起始鍵」集合。
formkeys	charset	「合法的公式起始鍵」集合。
cmdkeys	charset	「合法的命令起始鍵」集合。

legalkeys	charset	上述 4 個集合的聯集。
cmdchars	charset	「/」命令字元所構成的集合。
formchars	charset	「合法的公式字元」集合。

宣告的最大數量和 **Pascal** 所需的模組，不允許使得主要常式的邏輯弄得複雜。在全盤變數設定初值之後，初 Sheet 就被設定和顯示，**Pascal** 就進入一個環（loop）、它接受使用者鍵入的字元，也根據該字元的等級，而有所行動。它的動作有 4：**movecursor** 常式搬移游標到 sheet 上相鄰的 cell。**getlabel** 從使用者那接受一個標記。**getformula** 接受一個公式。**docommand** 實行表 8-1 中的任一命令。該環不斷的進行、直到 **alldone** 的旗標變成 **TRUE** 為止；而這個旗標是 P 伴隨 **Quit** 命令而產生的。

全盤變數定初值 (Initializing Global Variables)

Pascal 利用一個分離的常式，去將整個程式的變數命令初值化，常式設計如下：

```
segment procedure initpc;
( Initialize global variables )

var
    i: integer;

begin ( initpc )
    crt(CLEAR);
```

```

inpline := MAXCRTROW - 2;
statline := MAXCRTROW - 3;
center('Initializing...', statline);
alldone := FALSE;
ten[0] := 1;
for i := 1 to REGSIZE do
    ten[i] := 10 * ten[i - 1];
{$r-}
fillchar(dashes[1], MAXWIDTH, '-');
dashes[0] := chr(MAXWIDTH);
{$r+}
nrows := MAXCRTROW - YORG - 3;
sheetfmt.fmt := FIXEDPOINT;
sheetfmt.width := 10;
sheetfmt.ndigs := 2;
for i := 1 to MAXCOLS do begin
    colptr[i] := NIL;
    usesheetfmt[i] := TRUE;
end;
for i := 1 to MAXROWS do
    rowptr[i] := NIL;
autocalc := TRUE;
colcalc := TRUE;
zero.frac := 0;
zero.expo := MINEXP;
upkey := chr(15);
downkey := chr(12);
leftkey := chr(8);
rightkey := chr(21);
cursorkeys := [upkey, downkey, leftkey, rightkey];
labelkeys := ['A'..'Z', 'a'..'z', '"'];
formkeys := ['0'..'9', '+', '-', '.', '(', '['];
cmdkeys := ['/', '!', '>'];
legalkeys := cursorkeys + labelkeys + formkeys + cmdkeys;
formchars := formkeys + ['A'..'Z', '*', '/', ']', '!', ':', ' '];
cmdchars := ['C', 'E', 'F', 'I', 'L', 'M', 'P', 'Q', 'S', 'T'];
fmtchar[FIXEDPOINT] := 'F';
fmtchar[SCIENTIFIC] := 'S';
boochar[TRUE] := 'Y';
boochar[FALSE] := 'N';
statchar[OK] := ' ';
statchar[OVERFLOW] := 'O';
statchar[UNDERFLOW] := 'U';
statchar[ZERODIVIDE] := 'Z';
pagelen := 66;
lmarg := 10;
rmarg := 70;
printx1 := 1;
printx2 := 6;
printy1 := 1;
printy2 := pagelen - MTOP - MBOT - 1;
pinit := '';
pterm := '';
eraseline(statline)
end;

```

initpc中唯一不尋常的事是其第一線，**segment procedure initpc**；在APPLE 和UCSD PASCAL，可能設計成爲一個「段」。除非該常式被呼叫、否則對應一段常式的代碼，不會自磁碟載入記憶體。本常式執行完畢後、就被遺忘，代碼所占用的記憶空間，還給系統讓其他人使用。

initpc 設計成一段，是個很好的選擇，它只有在程式執行的開始被呼叫一次，所以如保留這一段、直到程式執行完畢就太浪費記憶空間了，如你的PASCAL 版本沒有這種選擇項目，只要在代碼中止該 **segment**字，就可以了。

游標移動鍵被APPLE II PASCAL 編修程式定一次初值；這些值隨機種不同而有不同的使用方法。

Sheet 顯示常式

Pascalc 利用若干常式來控制在CRT 螢幕上顯示Sheet。在 **initpc** 之後，第一個被呼叫的常式是 **drawsheet**，它顯示一個「概略」的Sheet；它描繪未被標記的水平 and 垂直的軸和一個空白游標位置指示器。

```
procedure drawsheets;
( Draw sheet axes )

var
  i: integer;

begin ( drawsheet )
  crt(CLEAR);
  posstr(dashes, XORG, YORG - 1);
  gotoxy(XORG - 1, YORG);
  for i := 1 to nrow do begin
    if i mod 5 = 0 then
      write('-')
    else
      write('!');
    crt(LEFT);
    crt(DOWN)
  end;
  posstr('E...J', 0, 0)
end;
```

在每 Sheet 上的第五行上選作標記；它由 **labelrows** 常式處理之。**labelrows** 有一個參數即列數，它被放在 Sheet 的最上一列。

```
procedure labelrows(r1: integer);
( Label rows )

var
  r, y: integer;

begin ( labelrows )
  firstrow := r1;
  lastrow := firstrow + nrow - 1;
  r := firstrow + 4;
  y := YORG + 4;
  while r <= lastrow do begin
    gotoxy(0, y);
    write(r: 3);
    r := r + 5;
    y := y + 5;
  end
end;
```

呼叫 **labelrows** 以便修訂 **firstrow** 和 **lastrow** 兩個全盤變數，它用來顯示現行所在的第一和最後的一列。

labelcols 程序與「將 Sheet 的行標記」工件類似，因為每行 Sheet 行占用 CRT 的好幾行，於是這個任務就變得更複雜。

```
procedure labelcols(c1: integer);
( Label columns )

var
  c, crtcol, wid: integer;

begin ( labelcols )
  posstr(dashes, XORG, YORG - 1);
  firstcol := c1;
  c := firstcol;
  repeat
    crtcol := colpos[c + 1] - colpos[firstcol] + XORG - 1;
    if crtcol <= MAXCRTCOL then begin
      posstr('!', crtcol, YORG - 1);
      lastcol := c;
      wid := colpos[c + 1] - colpos[c];
      if wid > 2 then begin
        gotoxy(crtcol - (wid + 1) div 2, YORG - 1);
        write(c)
      end
    end
    c := c + 1
  until (crtcol > MAXCRTCOL) or (c > MAXCOLS)
end;
```

本程序藉著把 **dash** 字串放在螢幕上特別地方，把前面任意行的標記被壞。在每一 **sheet** 行的最後 CRT 行上以驚歎號註記。若一個行夠寬的話，就在差不多是中央的地方標上行數。

在 **labelcols** 裡使用 **colpos** 陣列含有工作表的每一行的相對「起點位置」，使用 **colpos** 陣列，可以簡化計算 CRT 的最前和最後之行。例如，若 **Sheet** 行 **c1** 在 CRT **x** 行，則 **sheet** 行 **c2** 在 CRT 的 **colpos[C2] - colpos[c1] + x** 行。

colpos 陣列由 **stepcolpos** 程序定初值：

```
procedure setcolpos;
( Set column positions from format widths )

var
  c: integer;

begin ( setcolpos )
  colpos[1] := 0;
  for c := 1 to MAXCOLS do
    if usesheetfmt[c] then
      colpos[c + 1] := colpos[c] + sheetfmt.width
    else
      colpos[c + 1] := colpos[c] + colfmt[c].width
  end;
```

calpos[1] 是隨意設定為 1 的，至於 **colpos** 的其餘陣列的計算，只需觀察兩相鄰陣列 **colpos[c]** 和 **colpos[c+1]** 即可，這也就是 C 行的寬度。這寬度可能是 **sheet — format** 寬度，也可能是 **column — format** 寬度。計算 **colpos** 元素的常式非常簡單，只是把相對的行寬度加上任意元素，每當缺設 **sheet format** 或任何行格式改變了，就呼叫 **colpos**，和 **setcolpos** 兩常式，以設定初值。

dispsheet 是用來顯示 **sheet** 的程序，它有兩個參數 **c1** 和 **r1**，分別是 CRT 窗的左上角的行和列坐標。這程序也常被主常式呼叫，以便設定初值；每當 **sheet** 的展示改變或搬移窗後 **sheet** 的其他部份也要顯示時，它也就被其他常式呼叫了。

```
procedure dispsheet(c1, r1: integer);  
( Display sheet )  
  
var  
    r: integer;  
  
begin { dispsheet }  
    if r1 <> firstrow then  
        labelrows(r1);  
    if c1 <> firstcol then  
        labelcols(c1);  
    for r := firstrow to lastrow do  
        disprow(r)  
end;
```

當窗子被搬移了 (C1 值被 firstcol 改變, 或 r1 值被 firstrow 改變, 或兩者皆被改變) dispsheet 就先將 sheet 的列和行重新標記。此後 sheet 的窗的部門就被一系列一列的顯示了。

disprow 程序處理在 CRT 上一個 sheet 列的顯示; 它所在的位置即與「以螢幕為窗映至 work-sheet」相似。disprow 有時候還需決定 cell 的展示字串是否在窗子出現, 這種決定有時非常複雜, 因為該字串可能出現, 被窗子的邊修剪其左邊或右邊、或是兩邊。

disprow 的虛擬碼 (pseudo-code) 像:

```
begin  
    calculate CRT row for display  
    erase old display on CRT row  
    check cell pointed to by row header pointer  
    while (cell exists) and (we haven't moved off right edge  
                                of screen)  
        calculate CRT column where display string should begin  
        if CRT column > MAXCRTCOL  
            we've moved off right edge of screen  
        else  
            check for clipping on left  
                (calculate index of first displayed string  
                                character)  
            check for clipping on right  
                (calculate index of last displayed string  
                                character)  
            if any part of string can be displayed  
                display it at correct position
```

```

        end-else
        check next cell in row
    end-while
end

```

本虛擬碼譯成 PASCAL 的常式是：

```

procedure disprow(r: integer);
{ Display single row }

var
    cp: cellptr;
    done: boolean;
    crtrow, crtcol, c1, c2: integer;

begin { disprow }
    crtrow := YORG + r - firstrow;
    gotoxy(XORG, crtrow);
    crt(ERASEOL);
    cp := rowptr[r];
    done := FALSE;
    while (cp <> NIL) and not done do begin
        with cp^ do begin
            crtcol := colpos[cellcol] - colpos[firstcol] + XORG;
            if crtcol > MAXCRTCOL then
                done := TRUE
            else if display <> NIL then begin
                c1 := max(1, XORG - crtcol + 1);
                c2 := min(length(display^), MAXCRTCOL - crtcol + 1);
                if c2 >= c1 then
                    posstr(copy(display^, c1, c2 - c1 + 1), crtcol + c1 - 1,
                                                                    crtrow)
            end
        end;
        cp := cp^.rightptr
    end
end;

```

變數 **c1** 掌握第一個被顯示的字串的索引，而 **c2** 掌握最後一個被顯示的字串之索引。

disprow 使用 APPLE 和 UCSD PASCAL 的內建函數 **copy function**。

```
substr := copy(s, i, n);
```

自 S 字串第 **i** 個字元，抄 **n** 個字元到 **substr**。

disprow 同時還呼叫 **min** 和 **max**；這兩函數是非常簡單而且有用的常式，它們分別回轉一個較小或較大的整數值，該兩常式分述如下：

• 高等Pascal 程式設計技巧 •

```
function min(i1, i2: integer): integer;
  ( Return lesser of two integers )

begin ( min )
  if i1 <= i2 then
    min := i1
  else
    min := i2
end;

function max(i1, i2: integer): integer;
  ( Return greater of two integers )

begin ( max )
  if i1 >= i2 then
    max := i1
  else
    max := i2
end;
```

游標的放置與搬移 (Cursor Placement and Movement)

下一個主要設計的常式是處理游標：把一特定位置移到另一位置。

setcursor 程序，根據行或列坐標，在 cell 上放置游標：

假如，cell 現在的位置無法在螢幕上顯示，**setcursor** 就捲動窗，以便顯示之。捲動方式，有時是調整第一個被顯示的 sheet 行、或 sheet 列，有時同時調整。調整過程中唯一複雜的部份是，當游標自舊窗向右方移走；我們就必須去求 **firstcol** 的新值。於是呼，新游標 sheet 行就變成螢幕上顯示 sheet 的最後一行。**setcursor** 就根據 **colpos** 陣列的新的 sheet 行描繪之。

setcursor 於是在左上角修正游標位置指示器，以反映新游標位置。如 cell 的游標不為「空」，**setcursor** 就在窗下程式狀況線顯示 cell 的內容。該常式藉 **findcell** 函數、得知 cell 是不是「空」。

```
procedure setcursor(c, r: integer);
  ( Put cursor at specified spot )

var
  c1, r1: integer;
```



```

begin ( setcursor )
  if c < firstcol then
    c1 := c
  else if c > lastcol then begin
    c1 := c;
    while (colpos[c + 1] - colpos[c1] <= MAXWIDTH) and (c1 > 1) do
      c1 := c1 - 1;
    if colpos[c + 1] - colpos[c1] > MAXWIDTH then
      c1 := c1 + 1
  end
  else
    c1 := firstcol;
  if r < firstrow then
    r1 := r
  else if r > lastrow then
    r1 := r - rows + 1
  else
    r1 := firstrow;
  if (r1 <> firstrow) or (c1 <> firstcol) then
    dispheet(c1, r1);
  gotoxy(1, 0);
  write(c: 2);
  crt(RIGHT);
  write(r: 3);
  curscol := c;
  cursrow := r;
  curcp := findcell(curscol, cursrow);
  eraseline(statline);
  if curcp <> NIL then
    with curcp do
      if (fp = NIL) and (display <> NIL) then
        posstr(concat('Label:', display^), 0, statline)
      else if fp <> NIL then
        if fp^.formula <> NIL then
          posstr(concat('Formula:', fp^.formula^), 0, statline);
  gotoxy(XORG + colpos[c] - colpos[firstcol], YORG + r - firstrow)
end;

```

setcursor 完成之後，再設計一個搬動游標的常式就非常容易了：

根據使用者所定的字元型態，movecursor 搬移 sheet 游標，搬移時呼叫 setcursor，把游標放到鄰近的 cell。如使用者，想把游標搬移到 sheet 之外，則movecursor 程序就發出「嗶」的警告聲。

```

procedure movecursor(ch: char);
( Move cursor in direction specified by user )

begin ( movecursor )
  if (ch = upkey) and (cursrow > 1) then
    setcursor(curscol, cursrow - 1)
  else if (ch = downkey) and (cursrow < MAXROWS) then
    setcursor(curscol, cursrow + 1)
  else if (ch = rightkey) and (curscol < MAXCOLS) then
    setcursor(curscol + 1, cursrow)

```

• 高等 Pascal 程式設計技巧 •

```
else if (ch = leftkey) and (curscol > 1) then
    setcursor(curscol - 1, cursrow)
else ( attempt to move off sheet )
    crt(BEEP)
end;
```

遞增測試 (Incremental Testing)

測試一個程式，就把它當作是份內事，目的就在於找到錯誤，除錯、發現缺點，遞增測試的好處如下：

- 測試程式的一部份比測試整個程式容易測試一個長而且不是很有趣的過程，要保證其編譯正確，而且在不同的輸入，其運作正常。偶爾測試一個大程式，往往會忽略其一、兩個特性。測試程式可以管理的片斷就比較嚴密和完整。

- 當整個程式發展完成除錯比較容易。

遞增測試允許程式設計師，在剛設計好的獨立部份，留有錯誤，暫時擱置測試、等到整個程式完成了，再測試比較容易更正錯誤。

- 設計的缺點盡可能的容易偵測出。

就和除錯一樣，程式留有小缺點，在與程式合併以後容易。立刻修改比較容易。例如，我們發現 **Pascal** 的游標移動的方法笨拙，在設計其餘部份時立刻更改。

- 及早讓程式工作。

程式設計師的士氣也值得考慮。

基於上述觀點，我們已設計了足夠 **Pascal** 測試，只要再設計簡單的常式，供其他程序被 **Pascal** 的主常式呼叫時，編譯本程式，運轉本程式，以期保證最初 sheet 如期望一般顯示，游標的移動也很正確。

鍵入標記 (Entering Labels)

現在 **Pascale** 可以顯示一個 **sheet** (最起碼是一個空的)
，而且把游標搬移到任何 **cell** 的周圍，及時設計一個程序

getlabel 自使用者那取得一個標記，並且插入該 **sheet**。

getlabel 程序如下：

```
procedure getlabel(ch: char);
{ Get label from user }

var
  s: string;
  cp: cellptr;

begin { getlabel }
  if ch <> '' then
    ungetkey(ch);
  eraseline(inpline);
  posstr('Label:', 0, inpline);
  getstring(s, MAXCROW - 6, 6, inpline, '', [' '..'^'], FALSE);
  eraseline(inpline);
  if s <> '' then begin
    if curcp <> NIL then
      erase cell(curcp);
    if not storecell(curscol, cursrow, curcp) then
      memout
    else if not storestr(s, curcp^.display) then
      memout
  end;
  disprow(cursrow);
  setcursor(curscol, cursrow)
end;
```

通常 **getlabel** 自使用者那，取得一個標記之後，就在記憶體內儲存之。儲存方式是在該 **sheet** 稀疏矩陣裡產生一個新的 **Cell**。(若已存有一個非空 **cell**，在存新 **cell** 之前，就先把舊的 **cell** 清除掉。)。此後，再利用 **disprow** 顯示該新的標記。

如果已無足夠空間，去存新的 **cell** 或新標記字串，**getlabel** 就呼叫 **memout** 程序，把事實告訴使用者，因本常非常簡單，首先設計之：

```
procedure memout;
{ Report lack of memory }

begin { memout }
  remark('OUT OF MEMORY')
end;
```

未輸入 (Un-Input)

記得 **Pascal** 的主常式發現，使用者要找一個字母或一個雙括號，而需要鍵入一個標記。如使用者鍵入一個字母，該字母就被當作標記的第一個字元，然而問題是我們要利用 **getstring** 自使用人那取得一個標記，而 **getstring** 並不能取得標記的第一個字元；該字元已由鍵盤讀入。

解決本問題的方法很多，我們可以重新寫一個 **getstring**，當它取得起始輸入字串時，就當作一個輸入參數，但是完全重寫一個常式，來附加這個不重要的特性，似乎是浪費。

我們的方法是，在 **getlabel** 裡呼叫 **ungetkey**，所謂 **ungetkey** 常式與 **getkey** 相反。例如，呼叫

```
ungetkey(ch);
```

把字元 **ch** 「放回輸入緩衝器內」。換言之，下次 **getkey** 被呼叫時，它立刻回轉 **ch** 給呼叫常式，而不是等候使用人的輸入。當呼叫 **getkey** 時，它通常就以「鍵盤輸入」回轉之。

在尋找鍵盤輸入時，利用 **getkey** 核對一個「輸入緩衝器」，而 **getkey** 需要修正，修正的方法也很簡單。換句話說，輸入緩衝器被定義成全盤變數，它「讀」和「寫」指標到緩衝器內，它定義方式如下：

```
<var>
  inbuf: packed array [0..BUFSIZE] of char;
  readptr, writeptr: 0..BUFSIZE;
```

inbuf 是一個掌握字元等候去讀的陣列。**readptr** 這個變數指示「自緩衝器讀入的最後一個元素」。**writeptr** 寫到緩衝器的最後一個元素。如果這兩個指標相等，則沒有字元待讀到輸入緩衝器內。

每個程式利用這個新方法，都會將指標定初值：

```
...  
readptr := 0;  
writeptr := 0;  
...
```

ungetkey 常式就如下：

```
procedure ungetkey(ch: char);  
{ Put character into input buffer }  
  
begin { ungetkey }  
  newptr := (writeptr + 1) mod (BUFSIZE + 1);  
  if newptr <> readptr then begin  
    inbuf[newptr] := ch;  
    writeptr := newptr  
  end  
  else { buffer is full }  
    crt(BEEP)  
end;
```

其次，getkey 常式必須修飾，以便核對緩衝器內的一個字元（在核對鍵盤輸入之前），如下：

```
function getkey(var ch:char;valid:charset;shiftlock:boolean);  
{ Get valid key typed at keyboard or present in input buffer }  
var  
  ok: boolean;  
  
begin { getkey }  
  repeat  
    if readptr <> writeptr then begin  
      readptr := (readptr + 1) mod (BUFSIZE + 1);  
      ch := inbuf[readptr]  
    end  
    else begin  
      read(keyboard, ch);  
      if eoln(keyboard) then  
        ch := chr(13)  
    end;  
    if shiftlock and ...  
  
    { as before }  
    ...  
  end;
```

在 getkey 和 ungetkey 兩程序的模 (mod) 運算，使得 inbuf [BUFSIZE] 被使用之後，inbuf [0] 製造下一個元素，於是產生一個圓形或環狀緩衝器，把 inbut 重新安

排，把它圍成圓圈，`inbuf[BUFSIZE]` 與 `inbuf[0]` 相鄰，則有助於思考 `inbut` 的元素。

緩衝器只要有足夠的空間，我們就能釋放整個字元的字串，而這項功能由 `ungetstr` 施行之。

```
procedure ungetstr(s: string);
{ Put string into console input buffer }

var
  i: integer;

begin { ungetstr }
  for i := 1 to length(s) do
    ungetkey(s[i]);
end;
```

這個方法優點是實行方便並具易傳性，缺點是要修飾 `getkey`，不幸的是，我們曾以分開的方式對於 `getkey` 去編譯，而如今要修飾 `getkey`，就必須對這個分開編譯單元，重新編譯。所以，我們最好設計一個 `ungetkey` 常式，而不去修飾 `getkey`。

最起碼可以利用 APPLE PASCAL 寫這麼一個常式，利用 APPLE PASCAL 的作業系統 (Operation csystem) 在記憶裡有個固定位置，儲存由鍵盤輸入的字元。(位置是十六進位的 3 B 1，十進位的 945)，當使用者鍵入一個字元，作業系統就以上述位置做為緩衝器，將之儲存。當 APPLE PASCAL 利用 `read` 指令，去接受鍵盤字元，直接到緩衝器去取。該緩衝器名為「鍵前」緩衝器。

APPLE PASCAL 的鍵前 (type-ahead) 緩衝器，和 `inbuf` 陣列一樣，它是一個可容納 79 個字元的環式緩衝器，本系統的緩衝器由指標讀進，或寫出。讀指標是一個元組位址 BF18 (十六進位)，48920 (十進位)，而寫指標為 BF19 (十六進位)，48921 (十進位)。

假如我們可以接達 (access) 這個鍵前緩衝器，進而讀寫指標，我可寫一版的 **ungetkey**，它重複作業系統的「把一個字元放進緩衝器」動作，把它寫成組合語言是非常容易的任務，而我們用 PASCAL 來寫。

要切記的是，此時我們的程式，不具易傳性，**ungetkey** 代碼只能在 APPLE PASCAL 1.1 版下使用。只要不同版，即使是 APPLE PASCAL 亦不能通用。通常，使用其他版本的 APPLE PASCAL 必須使用「易傳性的」**ungetkey**，並且修飾 **getkey**。

首先，利用 PASCAL 寫一個 **Peek** 和 **Poke** 常式，你可重新檢討，組合，改用 BASIC 設計之。**Peek** 是用來檢查所賦予的記憶位置。其函數是。

```
membyte := peek(addr)
```

回轉一個位元組（位址是 **addr**），並把該值指派給變數 **membyte**。此外，**Poke** 是用來改變所予位置的值，程序呼叫方式是：

```
poke(addr, membyte);
```

把記憶位址 **addr** 的值改成 **membyte**。

通常，**peek** 和 **poke** 無法使 PASCAL 設計成具有易傳性，它簡直沒有辦法利用位址，接達記憶位置，我可利用「設定一個指標變數給記憶位置」，來改變或檢查記憶位址。

在大部份的 PASCAL 版本，一個指標變數就只包含位置。APPLE PASCAL 的指標，是一個工位元組、無正負符號的整數，它是以定址 65,536 個位元組。但，我們不能單單把位址指派給指標：整數和指標的資料型態不同，在 PASCAL 的文法裡，型態不同的資料不能相互指派。

大部份的 PASCAL 版本裡，可以用不同的記錄型態，把一個指標變數指定給一個所賦予的位址，首先考慮下面兩個資料的宣告：

```
<type>
  byte = 0..255;
  twobytes = packed array [0..1] of byte;
```

一個 byte 變數範圍是自 0 到 255，twobytes 是一個 2 位元組的被包裝的陣列。

下個步驟，是宣告一個不同的記錄型態，trixrec，這個型態包括一個整數，或一個指標，指向 twobyte 型態變數，如下：

```
<type>
  trixrec = record
    case boolean of
      TRUE:
        (ptr: ^twobytes);
      FALSE:
        (adr: integer)
    end;
```

對於 PASCAL 而言，不同的記錄型態，是節省記憶的方法；不同的變數型態可以用相同的記憶空間，如我們定義一個變數的型態 trixrec 如下：

```
<var>
  trix: trixrec;
```

則 trix 包括一個整數，或一個指向一個 twobyte 變數的指標：trix adr 為一整數，而 trix ptr 是一個指標，關鍵在於兩者參照相同的記憶範圍，它們占用相同的兩個位元組。

使用不同的記錄型態結構，peek 和 poke 就容易撰寫。對

於 **peek**，我們把變數型態 **trix** 定初值，使它包含一個整數位址，這個位址，是我們利用位址，指派到 **trix-addr** 去的。此時，指標 **trix-ptr** 指向記憶位置，利用 **trix-ptr[0]**，來接受記憶位置的內容。

```
function peek(addr: integer): byte;
{ Basic-like peek function }

var
  trix: trixrec;

begin { peek }
  trix.adr := addr;
  peek := trix.ptr[0]
end;
```

對於 **Poke**，以記憶位置，來設定 **trix** 變數的初值，再據之改變 **trix-ptr**。

```
procedure poke(addr: integer; val: byte);
{ Basic-like poke }

var
  trix: trixrec;

begin { poke }
  trix.adr := addr;
  trix.ptr[0] := val
end;
```

trixrec 記錄結構又稱為「自由型態——結合」(**free type-union**)，程式在執行期間，**PASCAL** 無法判別在 **trixrec** 變數裡儲存的是一個整數，或是一個指標。

有了 **poke**，就可立刻撰寫 **ungetkey**：

```
procedure ungetkey(ch: char);
{ Put character into console input buffer }

const
  RPTR = -16616;      { address of console buffer read pointer }
  WPTR = -16615;      { address of console buffer write pointer }
  CONBUF = 945;        { address of console buffer }
  CBUFLEN = 78;        { length of console buffer - 1 }
```

```

type
  byte = 0..255;
  twobytes = packed array [0..1] of byte;
  trixrec = record
    case boolean of
      TRUE:
        (ptr: ^twobytes);
      FALSE:
        (adr: integer)
    end;

var
  newp: byte;

{-----}
{ Modules to be inserted here: }
{      peek      }
{      poke      }
{-----}

begin { ungetkey }
  newp := (peek(WPTR) + 1) mod CBUFLEN;
  if newp <> peek(RPTR) then begin
    poke(WPTR, newp);
    poke(CONBUF + newp, ord(ch));
  end
  else { buffer is full }
    crt(BEEP)
end;

```

宣告的常數 **RPTR** 和 **WPTR**，分別代表「控制台——緩衝器」的位址，讀指標和寫指標。其有以負整數代表位址的必要；在 **APPLE PASCAL** 中，在記憶裡以有符號的 16——位元整數表示一個整數，而記憶位址却是沒正負符號的 16——位元整數。唯一不同之處是在位址大於 32767 時才會發生；在這種情況下，就以下列公式，把無正負符號的整數，換算成有正負符號的整數：

$$\text{<signed integer>} := \text{<unsigned integer>} - 65536$$

動態記憶配置 (Dynamic Memory Allocation)

下步驟要解決的是：如何儲存不同長度（如字串）的變數

，而不浪費記憶。理想狀況是，需要多少記憶體，就有多少記憶體；當不需要這些記憶體時，就還給系統以供他用，這種處理就叫動態記憶配置 (Dynamic Memory Allocation)，所謂「動態」是根據程式運轉需要才使用之意。

標準 PASCAL 幾乎提供我們之所需，假定變數 **sp** 是 **stringptr** 型態，則只要呼叫標準 PASCAL 的內建常式 **new**，配置記憶給一個字串，並指派記憶位址，以指向變數 **sp**。

標準 PASCAL 也提供一個解除被 **new** 配置的記憶，亦即 **dispose(sp)**；常式。

有許多 PASCAL 的實行中，瞬間儲存器 (storage) 可被 **sp** 指向，以讓出空間，還給系統，作為其他用途。

這個方法雖然並沒解決我們的問題，却也相去不遠，理由有二。第一，**new** 配置固定大小的記憶體；就如前面敘述，呼叫 **new(sp)** 就能保留儲存「80 一字元」字串的空間，說不定我真正所需是小於 80 字元。

其次，APPLE PASCAL 並未提供 **dispose** 函數，就算我們可以有效的分配變動——大小儲存區域，APPLE PASCAL 也無能為力去釋放空間，供我們使用。

我們將設計 **alloc** 和 **free** 來配置或取消配置記憶。當呼叫：

```
addr := alloc(nbytes);
```

發現記憶體的連續位元組 **nbytes**，並配置它們以供使用。**alloc** 回轉該區記憶體的第一個位元組的位址，若 **alloc** 找不到足夠的記憶空間，就回轉 0 表示失敗。

由 **alloc** 配置記憶體，就可由 **free** 將之解除之，下列呼叫

```
free(addr, nbytes);
```

解除自位址 **addr** 起 **nbytes** 位元組，以供再使用。**alloc** 和 **free** 不過是範例：如未經 **alloc** 配置一個位址，或者自配置過的址使用不同的 **nbytes** 值，是最危險不過的了。（在了解 **alloc free** 的工作狀況後，你就知道解偶爾使用上述的危險性。）

記憶體管理資料結構 (memory Management Data structure)

爲了 **alloc** 完成它的工作，我們必須發明「註記」尚未使用的記憶體的方法；否則 **alloc** 無法分辨，自由和被保留的記憶體。通常，自由記憶體爲一數量不等的不同大小之塊 (**block**) 所破壞，也與正在使用的塊同時改變。**alloc** 透過那些自由記憶體，去尋找足夠的空間，保持環式串列 (**circular list**)，我們也利用一些自由記憶空間本身儲存串列。

每一個自由記憶空間塊包括(1)一個整數——塊的大小，(2)一個指標鏈接 (**chain**) 下一塊，和(3)自由空間剩餘的部份。包含「塊的大小」和「指向下一塊的指標」的欄位稱之爲塊的標題 (**block header**)，每一自由塊的標題占用 4 個位元組。該結構詳見圖 8-3。自由串列有 4 個自由塊，分別是 500 位元組、344 位元組、208 位元組、和 1012 位元組，每一標題大小均由標題——大小單元衡量；爲了求得該塊的位元組總數，就乘以 4（這種安排的先決條件是每一自由塊的長度是 4 的倍數）。

除了圖 8-3 的 4 塊之外，尚有一個只有一個大小欄爲 0 的標題，這標題稱爲列串標題 (**list header**)。

最後，在列串中定義一個指標，指向一些元素，本程式就藉該指標接達列串（詳圖 8-3）。

雖然圖 8-3 並沒有標明「自由列串」，我們就加諸於自由記憶串列，一個額外的限制：列串的塊按記憶的位置次序排列，列串標題在最低的記憶位置，而最高位置的指標，又指回列

串標題。

自由列串結構在PASCAL 裡很容易定義，首先可宣告結構如下：

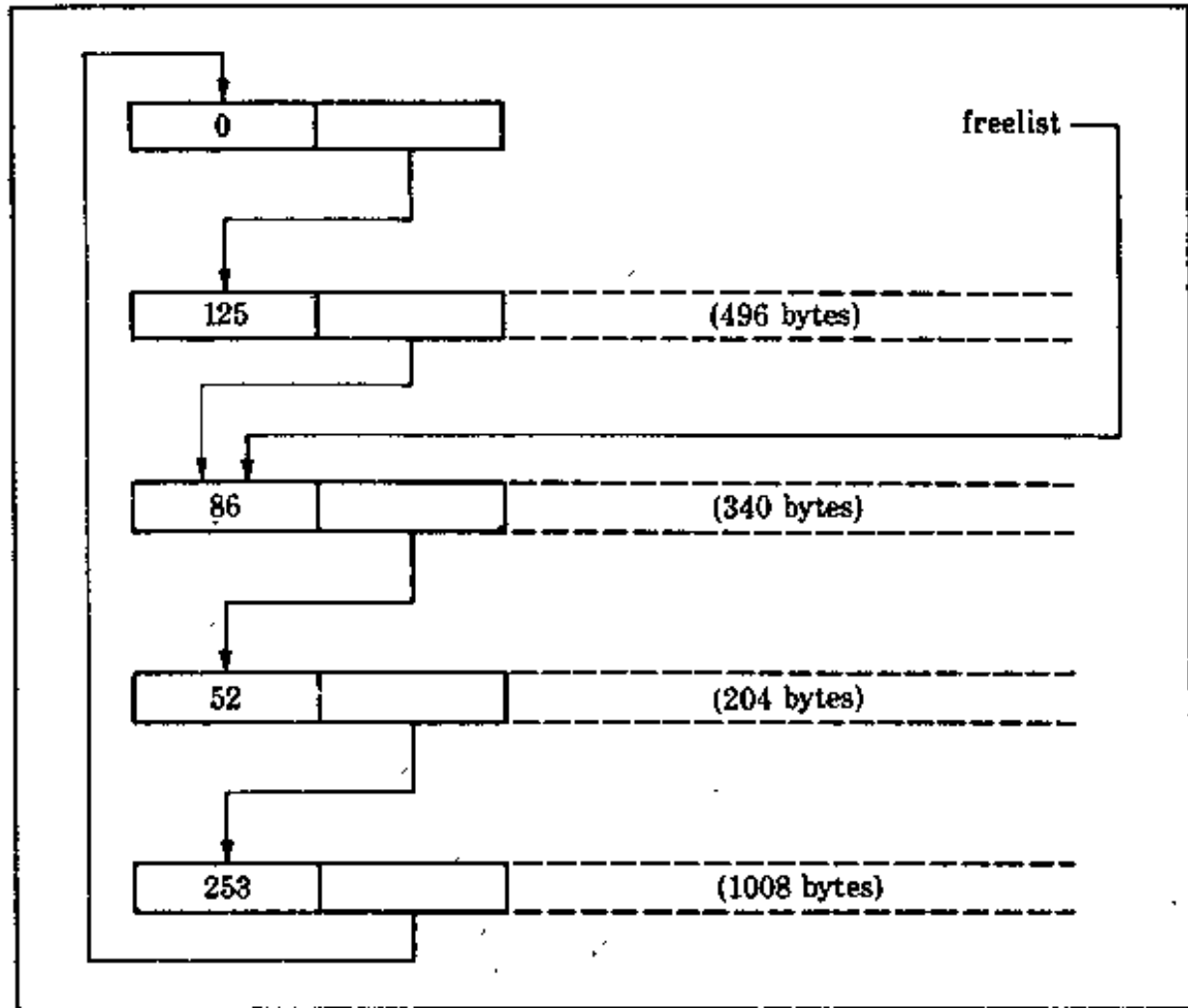


圖 8-3 自由記憶體列串資料結構

```
<type>
  headerptr = ^header;
  header = record
    size: integer;
    next: headerptr
  end;
```

我們同時定義指標，以指向列串，並視該列串為一個全盤變數如下：

```
<var>
  freelist: headerptr;
```

設計 **alloc** 過程中，我們假定自由記憶體列串，已由定初值常式設定了初值，而 **freelist** 指標指向任意自由記憶體的標題。於是，**alloc** 必須一個塊接一塊搜尋列串，以期找到一個足夠大的塊，容納這 **nbytes** 位元組，如能找到，就必須剩餘一些元組，以便接受從「自由列串」返回位址，才能呼叫本常式。當 **alloc** 離開自由串列時，必須有正確的大小欄位，和指標 **alloc** 如下：

```
function alloc(nbytes: integer): integer;
{ Find and allocate nbytes unused bytes of memory }

var
  lastblk, currbk: headerptr;
  nunits: integer;
  done: boolean;

begin { alloc }
  nunits := (nbytes + HDRSIZE - 1) div HDRSIZE;
  lastblk := freelist;
  currbk := lastblk^.next;
  done := FALSE;
  repeat
    if currbk^.size >= nunits then begin { success }
      if currbk^.size = nunits then begin
        lastblk^.next := currbk^.next;
        alloc := ord(currbk)
      end
      else begin
        currbk^.size := currbk^.size - nunits;
        alloc := ord(currbk) + currbk^.size * HDRSIZE
      end;
      freelist := lastblk;
      done := TRUE
    end
    else if currbk = freelist then begin { failure }
      done := TRUE;
      alloc := 0
    end
    else begin { try next block }
      lastblk := currbk;
      currbk := currbk^.next
    end
  until done
end;
```

因自由記憶塊以 4 個位元組的倍數儲存之，**alloc** 首先就必須把 **nbytes** 位元組的需求，翻譯成 **nunits** 標題——大小單元的需求，差不多也是 4 個位元組的倍數。（**HDRSIZE** 常數為 4、單位是位元組）。例如，**nbytes** 是 23，**alloc** 爲了要搜尋記憶塊，至少要有 6 個標題——大小單元，或 24 位元

組。

alloc 藉自由串列來搜尋，就要利用變數 **lastblk** 和 **currbk**。**lastblk** 在 **currbk** 之後，而 **currbk** 永遠指著前面一個元素 **lastblk**，而 **lastblk** 指向 **currbk**。

在搜尋過程中，**currbk** 所指向的塊之 **SIZE** 必須接受檢查，看看是否滿足 **nunits** 單元的需求，若夠大的塊被找到（**currbk · SIZE = nunits**），這個塊又從自由列串解除掛鈎，而該自由列串又為前一塊的 **next** 指標為指；**alloc** 回轉被解除掛鈎的位址。

如流程塊（**current block**）較所要求的大（**currbk . SIZE > nunits**），**alloc** 就被 **nunits** 減少「塊的大小」欄位；純益效應是切掉塊的最後 **nunits** 個單元，**alloc** 於是回轉初切除部份的地址。

如無法找到足夠記憶空間，則搜尋失敗。當 **currbk** 被包在搜尋的起點，而該塊又被 **freelist** 所指時，就會發生搜尋失敗。

alloc 中用到一個訣竅 **ord** 函數，該函數並非標準 **PASCAL** 的標準函數，只能在 **APPLE PASCAL** 和 **UCSD PASCAL** 中使用之。當 **ord** 被呼叫時（有一個指標變數作引數）回轉整數位址。

adr := ord(ptr)

回轉的位址由 **ptr** 所指，我們可以利用前面所提到的自由——聯（**free-union**）的技巧來描述之，但是，使用 **ord** 比較簡明。

alloc 設計完成了、**free** 又如何呢？通常，所有的 **free** 都應該回轉那「已自由的塊給「環式自由列串結構」。首先，已自由的塊的 4 個位元組必須轉換成一個塊標題，作為 **SIZE** 欄位之用。則該塊就必須和自由列串重新鍊接：它的 **next** 指標

欄必須指向下個自由塊，並且前一個自由塊的 next 指欄必須指著當前已自由的塊。

如果流程自由塊的相鄰兩邊中的任一邊是自由的，這兩塊就可以接合成一個較大的塊，若兩邊都已自由了，這三塊也可接合成一個塊。當不同大小塊，連續的重置和自由，則 Pascal 就可以更為節省空間。

free 使用下列策略：掃描自由列串，尋求適當地方，讓已自由的塊，在兩個流程塊中間或流程塊與一個 end 之間插入。當這個適宜的地方被找到了，該塊就被列串鍊接，並與相鄰的塊相接合，free 設計如下：

```
procedure free(addr, nbytes: integer);
{ Return block of memory to free list }
var
  done: boolean;
  prevblk, nextblk, freeblk: headerptr;
begin { free }
  moveleft(addr, freeblk, 2);
  freeblk^.size := (nbytes + HDRSIZE - 1) div HDRSIZE;
  prevblk := freelist;
  nextblk := prevblk^.next;
  repeat
    done := (ord(freeblk) > ord(prevblk)) and (ord(freeblk) < ord(nextblk));
    if not done then
      if ord(prevblk) >= ord(nextblk) then
        done := (ord(freeblk) > ord(prevblk)) or
          (ord(freeblk) < ord(nextblk));
      if not done then begin
        prevblk := nextblk;
        nextblk := nextblk^.next;
      end
    until done;
    if ord(freeblk) + freeblk^.size * HDRSIZE = ord(nextblk) then begin
      freeblk^.size := freeblk^.size + nextblk^.size;
      freeblk^.next := nextblk^.next;
    end
  else
    freeblk^.next := nextblk;
    if ord(prevblk) + prevblk^.size * HDRSIZE = ord(freeblk) then begin
      prevblk^.size := prevblk^.size + freeblk^.size;
      prevblk^.next := freeblk^.next;
    end
  else
    prevblk^.next := freeblk;
  freelist := prevblk;
end;
```


現在你當能明白，為什麼自由列串要按位址次序排列了：我們易於決定兩塊是否彼此相鄰。

free 使用了前面提到的 **ord** 函數之高度技巧，是爲了比較塊的位址和計算在記憶體裡的兩塊是否相鄰？此外，它又把傳給 **free** 的位址翻譯成一個塊標題指標。呼叫

```
moveleft(addr, freeblk, 2);
```

搬移兩個位元組的整數變數 **addr**，到標題指標變數 **freeblk**，使得 **freeblk** 指向位址 **addr**。**moveleft** 僅自記憶體的一點，搬兩個位元組到另一點上，搬移時並不檢查型態。**moveleft** 也可用相同的方法，把指標翻譯成位址。（我們可以使用 **moveleft** 而捨用 **peek** 和 **poke**）。

APPLE PASCAL 記憶體管理

我們只剩一個問題，即使用 **alloc** 和 **free** 把自由記憶體列串設定初值，第一個步驟，先調查在程式執行期間，電腦的記憶體被使用情形，才能很安全的設定自由記憶體列串。現在，就運用到 **APPLE PASCAL** 方面，詳加說明，其他版本的 **PASCAL** 用法與之相似。

程式在被執行的時候，其記憶被下列裝置使用：輸入／出緩衝器，程式代碼、資料瞬間儲存，作業系統、等等。這些對我們來說，都不重要，它們可以在有效的記憶體裡的任意位置。而 **APPLE PASCAL** 中，所有系統記憶體的使用是記憶體的底（低位址），或記憶體的頂（高位址）。它的範圍一定在程式裡的有效記憶體之內，所以其範圍與程式的大小息息相關。

在 **APPLE PASCAL** 裡，保有兩個自由記憶體區域，底端稱爲“heap”，頂稱爲堆疊（stack），詳見圖 8-4。

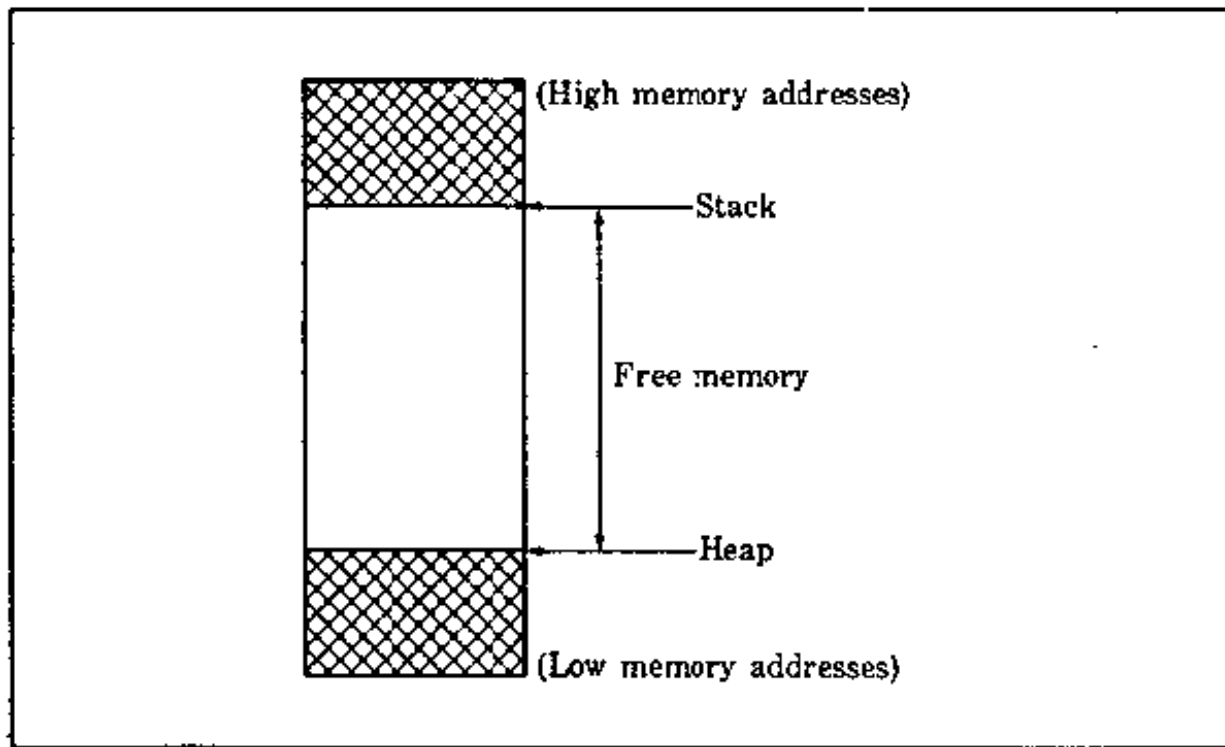


圖 8 - 4 APPLEPASCAL 記憶佈置

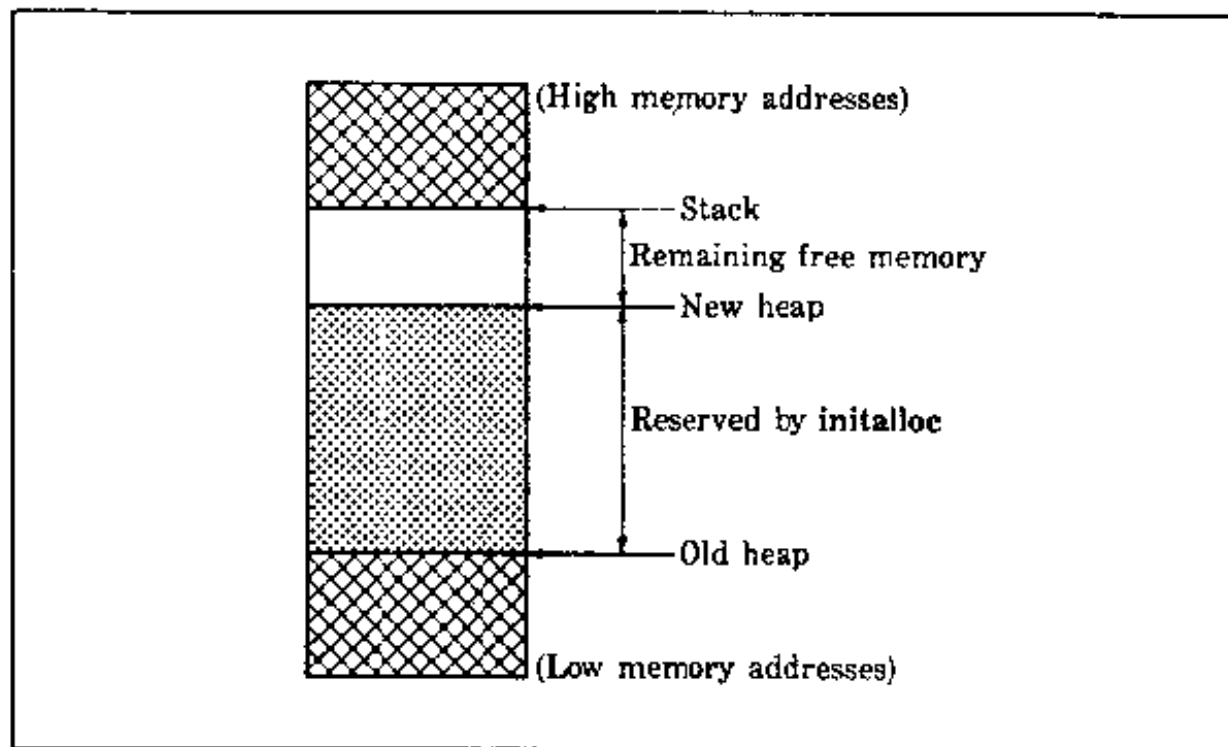


圖 8 - 5 自由串列設定初值後的記憶佈置

爲了呼叫常式，系統利用堆疊（stack），取得必要的暫時儲存器。每當一個程序（或一個函數）被呼叫，就必須有一定量的記憶，來儲存該程序的當地變數（local variables），回轉位址，和其他「簿記」資訊，所以每當一個程序被呼叫，堆

疊就降到記憶體的下方便，而且當程序的時候，記憶體恢復自由之際、堆疊又回到記憶體的上面。

而 heap 是當使用 new 時，用來配置動態變數：當 new 配置一個動態變數時，該 heap 移到記憶體的上面。

每當 heap 和 stack 兩者相碰撞，自由記憶體就被壓縮而沒有內容，程式就退出系統，並產生錯誤訊息：系統超出記憶體。

我們的戰略是，在程式開始的時候，設定自由記憶列串的初值時，保留一大塊記憶體，留相當小的自由記憶空間供系統呼叫常式，和其他用途。這一大塊一定要在起始 heap 的頂；於是 heap 重新配置該塊，此時的純效益，是該大塊受到保護，圖 8-5 是自由串列定初值，而改變記憶佈置圖。

接下來的問題是移動 heap：它可由「APPLE 和 UCSD PASCAL 的三個內建程序」完成之：mark，release，和 memavail。假定變數 hptr 已被宣告成一整數指標，如下：

```
<var>  
    hptr: ^integer;
```

則呼叫

```
mark(hptr);
```

設定變數 hptr 指向流程 heap，那也就是說，它包含 heap 的位址。另一方面，呼叫

```
release(hptr);
```

設定系統的 heap 指標，指向 hptr 裡的位址。簡言之，mark 允許我們去找 heap，而 release 改變它的地位。

內建函數 memavail 用來決定「在程式執行時，自由空間

的大小」。呼叫

```
freewords := memavail;
```

回轉在 heap 和 stack 之間的二個位元組的字。

一旦保留一個大塊，就能它裡面，設定自由串列，這些伴隨 **initalloc** 函數發生。只要呼叫

```
success := initalloc(leave);
```

就可以設定一個起始自由串列；成功時，回轉 **TRUE**，如無足夠空間，則回復 **FALSE** 引數 (argument) **leave** 是自由記憶體 **initalloc** 中 2 一位元組字的數量，它是用來讓系統保留呼叫常式之用。這個參數的設定，是防止在程式執行時發生堆疊溢位的錯誤。事實上，正確的設定本參數是一種實驗與錯誤。

initalloc 設計如下：

```
function initalloc(leave: integer): boolean;
{ Initialize dynamic allocation scheme }

var
  heap: ^integer;
  nunits, addr: integer;
  listhead, bigblock: headerptr;

begin { initalloc }
  nunits := (memavail - leave) div 2;
  if nunits < 2 then
    initalloc := FALSE
  else begin
    initalloc := TRUE;
    mark(heap);
    addr := ord(heap);
    moveleft(addr, listhead, 2);
    addr := addr + HDRSIZE;
    moveleft(addr, bigblock, 2);
    addr := addr + HDRSIZE * (nunits - 1);
    moveleft(addr, heap, 2);
    release(heap);
    with listhead^ do begin
      size := 0;
      next := bigblock
    end;
    with bigblock^ do begin
      size := nunits - 1;
      next := listhead
    end;
    freelist := listhead
  end
end;
```

本常式就像 **munits** 一樣，先計算要保留多大的塊，標題——大小單元的數量，根據流程 **memavail** 和留給系統記憶體的数量，計算出來的。最小塊是兩標題：一為串列標題，另一是其餘的串列標題。如沒有足夠的空間，容納這兩標題，**initalloc** 就向主常式報告失敗。否則，該 **initalloc** 呼叫 **mark** 和 **release** 把 **heap** 搬移到塊的上面，然後設立起始自由串列結構。它利用 **ard** 和 **moveleft** 去翻譯指標和整數型態。

最後要注意的是記憶——配置工具：許多版本的 PASCAL 和我們一樣擁有上述的內建常式。例如，UCSD PASCAL 的 IV.0 版，和下面要介紹的常式 **varnew** 和 **vardispose** 都是用來配置和自由化那些記憶體的不同大小之塊。這種情況下，使用內建常式，就比這裡陳列的方便了。

字串的動態儲字 (Dynamic Storage of Strings)

我們的動態記憶配置工具，通常不直接使用，而由 **alloc** 和 **free** 呼叫時，儲存或消除不同變數型態時使用之。例如，**storestr** 被用來在動態記憶體內儲存一字串。

```
function storestr(var s: string; var sp: stringptr): boolean;
( Store string in dynamic memory, return pointer to it )

var
  saddr: integer;

begin ( storestr )
  saddr := alloc(1 + length(s));
  if saddr = 0 then begin
    sp := NIL;
    storestr := FALSE
  end
  else begin
    moveleft(saddr, sp, 2);
    sp := s;
    storestr := TRUE
  end
end;
end;
```

這個常式利用 **alloc** 找和配置 **s** 字串的空間，則 **alloc** 需要 $\text{length}(s) + 1$ 個位元組：字串裡的字元數加 1。如沒有足夠的空間去儲存字串，**alloc** 就回轉 0，而 **storestr** 回轉 **NIL** 和函數結果 **FALSE** 給呼叫者，以表失敗。否則，它就在配置的地方，儲存字串，回轉一指標，指向字串，就如 **var** 參數 **sp** 和 **TRUE** 的函數結果。

erasestr 將儲存的字串，從動態記憶體內刪除：

```
procedure erasestr(sp: stringptr);  
( Erase string from dynamic memory )  
  
begin ( erasestr )  
  if sp <> NIL then  
    free(ord(sp), length(sp*) + 1)  
end;
```

erasestr 僅呼叫 **free**，回復那前面由 **alloc** 保留給自由列串的空間。

cell 儲存

我們現在設計一常式，自稀疏矩陣，儲存和刪除 **cells**，首先用 **storecell** 儲存 **cells**，呼叫

```
success := storecell(col, row, cp);
```

以便在記憶體內，建立一 **cell** 的行與列坐標 (**col** 和 **row**)。
var 參數 **cp** 回轉一指標給建立的 **cell**；函數結果為一布耳，以反映其是否很順利的被建立。

storecell 較 **storestr** 稍複雜，除了為 **cell** 配置空間之外，**storecell** 還要鏈接 **cell** 與稀疏矩陣結構。茲將 **storecell** 敘述如下：

```

function storecell(col, row: integer; var cp: cellptr): boolean;
( Create new cell at specified position, return pointer to it )

var
  celladdr: integer;
  cp1, cp2: cellptr;
  done: boolean;

begin ( storecell )
  celladdr := alloc(sizeof(cellrec));
  if celladdr = 0 then begin
    cp := NIL;
    storecell := FALSE;
  end
  else begin
    moveleft(celladdr, cp, 2);
    with cp^ do begin
      cellcol := col;
      cellrow := row;
      display := NIL;
      fp := NIL;
      cp1 := NIL;
      cp2 := colptr[col];
      done := FALSE;
      while (cp2 <> NIL) and not done do
        if cp2^.cellrow <= row then begin
          cp1 := cp2;
          cp2 := cp2^.downptr
        end
        else
          done := TRUE;
      downptr := cp2;
      if cp1 = NIL then
        colptr[col] := cp
      else
        cp1^.downptr := cp;
      cp1 := NIL;
      cp2 := rowptr[row];
      done := FALSE;
      while (cp2 <> NIL) and not done do
        if cp2^.cellcol <= col then begin
          cp1 := cp2;
          cp2 := cp2^.rightptr
        end
        else
          done := TRUE;
      rightptr := cp2;
      if cp1 = NIL then
        rowptr[row] := cp
      else
        cp1^.rightptr := cp
    end;
    storecell := TRUE
  end
end;

```

(Link into column)

(Link into row)

storecell 使用在 APPLE 和 UCSD PASCAL 都有效的函數 **sizeof**，在它的引數。回轉型態所需的儲存體（計算單位是位元組）。如你的 PASCAL 缺少這內建函數，你就無法寫一易傳性的 **sizeof** 的版本，只好用變數或型態來取代這函數了。對於你的 PASCAL 的實行方面，一定要有文件，並有充份的資訊，來決定這個儲存需求。

erasecell 常式，從稀疏矩陣和由 cell 配置的自由記憶體中，搬移（解除鏈接）一個特定的 cell，茲將 **erasecell** 列舉如下：

```
procedure erasecell(cp: cellptr);
{ Delete cell from sheet }

var
  cp1: cellptr;

begin { erasecell }
  if cp <> NIL then begin
    with cp^ do begin
      eraseform(fp);
      erasestr(display);

      { Unlink from column }
      if colptr[cellcol] = cp then
        colptr[cellcol] := downptr
      else begin
        cp1 := colptr[cellcol];
        while cp1^.downptr <> cp do
          cp1 := cp1^.downptr;
        cp1^.downptr := downptr
      end;

      { Unlink from row }
      if rowptr[cellrow] = cp then
        rowptr[cellrow] := rightptr
      else begin
        cp1 := rowptr[cellrow];
        while cp1^.rightptr <> cp do
          cp1 := cp1^.rightptr;
        cp1^.rightptr := rightptr
      end

      end;
      free(ord(cp), sizeof(cellrec))
    end
  end;
end;
```


此外，`erasecell` 呼叫 `erasestr` 去刪除 `cell` 的顯示字串，而 `eraseform` 刪除 `cell's` 的公式資料。

公式儲存體 (Formula Storage)

最後以儲存和刪除資訊記錄為例。這兩個都很容易完成。
`storeform` 為一 `forminfo` 記錄，配置空間，並且回轉一個指標。`storeform` 設計如下：

```
function storeform(var fp: formptr): boolean;  
{ Create formula info record }  
  
var  
    fadd: integer;  
  
begin { storeform }  
    fadd := alloc(sizeof(forminfo));  
    if fadd = 0 then begin  
        fp := NIL;  
        storeform := FALSE  
    end  
    else begin  
        moveleft(fadd, fp, 2);  
        fp^.cellval := zero;  
        fp^.usecolfmt := TRUE;  
        fp^.cellstat := OK;  
        fp^.formula := NIL;  
        storeform := TRUE  
    end  
end;  
end;
```

`eraseform` 呼叫 `erasestr` 去刪除該公式字串，再呼叫 `free` 回轉那 `forminfo` 記錄給自由記憶體，`eraseform` 如下：

```
procedure eraseform(fp: formptr);  
{ Delete formula info record from dynamic memory }  
  
begin { eraseform }  
    if fp <> NIL then begin  
        erasestr(fp^.formula);  
        free(ord(fp), sizeof(forminfo))  
    end  
end;
```

這常式可以在工作單上，插入標記，在處理下一步驟之前

, 最好先例試這些。

進入公式 (Entering Formulas)

下個步驟是設計 **getformula** , 自使用者取得公式, 計算的值, 和格式結果:

```
procedure getformula(ch: char);
( Get formula from user )

var
  s: string;
  success: boolean;

(-----)
( Modules to be inserted here: )
(      getformstr      )
(-----)

begin ( getformula )
  ungetkey(ch);
  eraseline(inplace);
  posstr('Formula:', 0, inplace);
  getformstr(s);
  eraseline(inplace);
  if s <> '' then begin
    if curcp <> NIL then
      erasecell(curcp);
    success := storecell(curscol, cursrow, curcp);
    if success then
      success := storeform(curcp^.fp);
    if success then
      success := storestr(s, curcp^.fp^.formula);
    if not success then
      menout
    else if autocalc then begin
      recalc;
      reformat;
      dispheet(firstcol, firstrow)
    end
    else begin
      calccell(curcp);
      formatcell(curcp);
      disprow(cursrow)
    end
  end;
  setcursor(curscol, cursrow)
end;
```

許多 **getformula** 已非常熟悉, 第一件事是呼叫 **ungetkey** 去放公式回到鍵——前緩衝器的第一個字母, 在接受公式, 視

```

procedure getformstr(var s: string);
{ Get pascal formula string }

const
  BS = 8;           { ASCII backspace key }
  CAN = 24;         { ASCII cancel key }
  ESC = 27;         { ASCII escape key }
  CR = 13;          { ASCII return key }
  DEL = 127;        { ASCII delete key }
  FLDCHR = '_';     { Input field marker }
  INPCOL = 8;       { Column where input starts }

var
  ch: char;
  okset: charset;
  relcol, relrow, i: integer;
  s1: string;

{-----}
{ Modules to be inserted here: }
{      getrc      }
{-----}

begin { getformstr }
  s := '';
  okset := formchars + [chr(BS), chr(CR), chr(CAN), chr(ESC), chr(DEL)];
  gotoxy(INPCOL, inpline);
  for i := 1 to MAXCRTCOL - INPCOL do
    write(FLDCHR);
  gotoxy(INPCOL, inpline);
  while getkey(ch, okset, TRUE) <> chr(CR) do
    if (ch in [chr(BS), chr(DEL)]) and (length(s) > 0) then begin
      crt(LEFT);
      write(FLDCHR);
      crt(LEFT);
      chopchar(s)
    end
    else if (ch = chr(CAN)) and (length(s) > 0) then begin
      for i := length(s) downto 1 do begin
        crt(LEFT);
        write(FLDCHR);
        crt(LEFT);
      end;
      s := '';
    end
    else if ch = chr(ESC) then
      getrc(INPCOL + length(s), inpline)
    else if (ch in formchars) and (length(s) < MAXCRTCOL - INPCOL) then
      begin
        addchar(s, ch, MAXSTR);
        write(ch)
      end
    else { Illegal character typed }
      crt(BEEP);
  for i := length(s) + 1 to MAXCRTCOL - INPCOL do
    write(' ');
end;

```

為字串 `s`，它被呼叫 `erasetcell` 刪除了游標位字的任何 `cell`，`getformula` 於是呼叫 `storecell`，在游標位置，建立一個新的 `cell`，並藉呼叫 `storeform` 為 `cell` 建立一個公式資訊記錄，最後它呼叫 `storestr` 來儲存公式字串。

任何一個動態記憶體配置都可能導致，超出記憶範圍的錯誤。於是，`getformula` 在每一次配置之後，都檢查、核對；在任何步驟發生任何錯誤，都由 `memout` 通知程式。

假如一切配置都成功，`getformula` 都根據布耳全盤變數 `autocalc` 來決定做一、二件事情。當它是 `TRUE`，`getformula` 重新計算，重定格式，重新展示全張工作單，否則 `getformula` 只影響已鍵入的 `cell`，計算已鍵入的式子，將其定格式，並在螢幕流程位置顯示之。

首先考慮，如何從用戶取得公式字串，可利用 `getstring` 去接受公式，就如同利用它，自 `getlabel` 取得標記。我們尚須加一個，在 `getstring` 內無效的特性。正如前面提到的，相對 `cell` 可用游標的移動鍵，指出其所參照的 `cell`（這個考慮，遠較「強迫使用人計算 `cells` 的變更方法」周詳得多。）

所以，我們利用 `getformula` 呼叫 `getformstr` 來輸入字串，而不用 `getstring`。`getformstr` 的設計，與 `getstring` 相似，茲比較兩常式如下：

這個新的代碼如下：

```
...  
else if ch = chr(ESC) then  
    getrccl(INPCOL + length(s), inpline)  
else  
    ...
```

`getformstr` 用 `<ESC>` 鍵，做為一信號，讓使用人藉按游標運動鍵，來鍵入一相對 `cell` 參考。`getformstr` 的動作像 `getstring` 核對和修訂，它呼叫 `getrc` 以處理游標的移動。

`getrccl` 在使用人面前呈現：`<ESC>` 鍵入表要進入一個

相對 cell 參考，而參考〔 + 0 , + 0 〕 在公式字串的流程入口 (current entry) 位置出現。CRT 的游標就被放在工作單的流程格子，依需要來按游標移動鍵；在公式字串的格子參考也據之調整。比如說，假如爲了格子，在【23,10】上鍵入一個公式，按〈ESC〉進入流程移動模式，則按向上移的鍵五次，向右二次，於是浮標就在【25,5】的位置出現，而公式字串，就會展示相對格子參考【+ 2 , - 5】。

當游標已放到希望放的格子，使用人再按〈ESC〉一次。該格子參考，就附加到已鍵入的公式之後，必要的話，工作單就在原來的窗重新顯示，使用人即可如前面提到的一樣，鍵入和編修公式。getrec 列舉如下：

```

procedure getrec(crtcol, crtrow: integer);
{ Get relative cell coordinates from user's cursor-movement commands }

var
  ch: char;
  len, relcol, relrow, oldrow1, oldcol1, oldcrow, oldccol: integer;
  s: string;
  okchars: charset;

{-----}
{ Modules to be inserted here: }
{      ungetstr      }
{-----}

begin { getrec }
  if crtcol > MAXCRTCOL - 10 then { no room }
    crt(BEEP)
  else begin
    okchars := cursorkeys + [chr(ESC)];
    oldrow1 := firstrow;
    oldcol1 := firstcol;
    oldcrow := cursrow;
    oldccol := curscol;
    len := 0;
    repeat
      relcol := curscol - oldccol;
      relrow := cursrow - oldcrow;
      ctos(relcol, relrow, TRUE, TRUE, s);
      posstr(s, crtcol, crtrow);
      if length(s) < len then
        write(' ');
      len := length(s);
      gotoxy(XORG + colpos[curscol] - colpos[firstcol],
              YORG + cursrow - firstrow);
      if getkey(ch, okchars, FALSE) in cursorkeys then
        movecursor(ch)
    until (ch = chr(ESC));
  end;
end;

```

```
until ch = chr(ESC);
if (firstrow <> oldrow) or (firstcol <> oldcol) then
  dispheet(oldcol, oldrow);
setcursor(oldcol, oldrow);
ungetstr(s);
posstr('_____', crtcol, crtrow);
gotoxy(crtcol, crtrow)
end
end;
```

一個相對格子參考，可以有 10 個字元（連同括號在內），第一件事，是 **getrec** 核對，是否有空間儲存這 10 個字元；如果沒有，就在發出嗶聲，並立刻返回 **getformstr**（如你已接受建議，修改 **getstiring**，允許字串的輸入，與螢幕邊界重疊，你也可做類似的改變 **getformstr** 和 **getrec**，也允許他們輸入，較長的公式。）

getrec 使用 **ungetstr**，把相對格子參考字串，放到鍵—前緩衝器，再被 **getformstr** 抄取，這總比一個參數傳輸，來得簡單。

ctos 程序，是被用來轉換一對座標成一個字串：

```
procedure ctos(col, row: integer; relcol, relrow: boolean; var s: string);
( Convert coordinate pair to string )

var
  s1: string;
begin ( ctos )
  s := '[';
  if relcol and (col >= 0) then
    addchar(s, '+', MAXSTR);
  itos(col, 0, s1);
  s := concat(s, s1, ',');
  if relrow and (row >= 0) then
    addchar(s, '+', MAXSTR);
  itos(row, 0, s1);
  s := concat(s, s1, ']');
end;
```

recol 和 **relrow** 兩個布耳參數，控制格子參考是相對的，抑或是絕對的。如果是 **TRUE**，該對應的座標，被轉換後，有一個引導符號。（注意，通常列或行座標可能是相對，也可能是絕對。）

重新計算 (Recalculation)

現在重新面對再計算的問題：在每一格子，和指派結果給格子一個值，評估公式，第一步驟是設計 **recalc**，它在工作表上，重新計算公式格子：

```

procedure recalc;
{ Recalculate sheet }

var
  r, c: integer;
  cp: cellptr;

begin { recalc }
  center('Recalculating...', statline);
  if colcalc then
    for c := 1 to MAXCOLS do begin
      cp := colptr[c];
      while cp <> NIL do begin
        calcCell(cp);
        cp := cp^.downptr
      end
    end
  else { recalc by row }
    for r := 1 to MAXROWS do begin
      cp := rowptr[r];
      while cp <> NIL do begin
        calcCell(cp);
        cp := cp^.rightptr
      end
    end
  end;
  eraseLine(statline)
end;

```

全盤布耳參數 **colcalc** 控制重新計算的次序，如 **colcalc** 是 **TRUE**，由行重新計算完成，使自【1,1】，其次【1,2】，再來是【1,3】，等等，再接著是【2,1】，【2,2】……，等等，如果 **colcalc** 是 **FALSE**，列重新計算，自左至右，完成第 1 列，第 2 列，……等等。

決定重新計算的次序，與工作表的邏輯結構有關；如果格子 A 與格子 B 相關，則格子 B，就得在格子 A 之前被計算。如賦予一格子（直接地或間接地）與其上或其右的(←)格子有關，

• 高等Pascal 程式設計技巧 •

就必須重新計算列，如所予的一格子，和其下或其左的格子相關，就重新計算行。

recalc 呼叫 **calccell** 評估一個單一格子公式，茲列舉 **calccell** 如下：

```

procedure calccell(cp: cellptr);
{ Recalculate cell }

var
  i: integer;

{-----}
{ Modules to be inserted here: }
{      evalexpr      }
{-----}

begin { calccell }
  i := 1;
  if cp <> NIL then
    if cp^.fp <> NIL then
      with cp^.fp^ do
        if formula <> NIL then
          cellstat := evalexpr(formula^, i, cellval);
end;

```

該 **calccell** 呼叫的 **evalexpr** 常式，與第 4 章 **calc** 所用的常式一樣。事實上，整個程式後面跟著相同模式，如 **calc**：在 **calc** 和 **pascalc** 內，**evalexpr** 巢式都包含有 **xsub** 和 **evalterm**。

我們可畫一巢狀圖形，來展式各別的模式，彼此在本節的程式內被包著。

```

calccell
  evalexpr *
    xsub *
    evalterm *
      xmul *
      xdiv *
      evalfact
        stox *
        findid *

```

(continued)

(continued)

```

getcoord
evalcell
evalid
findarg
evalsum
summatrix

```

這裡展式一些常式，在程式內是集狀的，**xsub** 和 **ewlterm** 被包在 **evalexpr** 內；**xmul**，**xdiv**，和 **evalfact** 包在 **evalterm** 內，等等。

圖內的星號跟在常式名字之後，顯示該常式抄自 **calc**，內容絲毫不改。第一個重新設計的是 **evalfact**，它必須被修正，以便識別格子參考和函數，**evalfact**：

```

function evalfact(var s: string; var i: integer; var x: xreal): xresult;
{ Evaluate factor }

var
  c: char;
  negfact: boolean;
  id: identifier;

{-----}
{ Modules to be inserted here: }
{      getcoord      }
{      evalcell      }
{      stox          }
{      findid        }
{      evalid        }
{-----}

begin { evalfact }
  negfact := FALSE;
  if gnbchar(s, i, c) in ['+', '-'] then begin
    i := i + 1;
    negfact := (c = '-')
  end;
  if gnbchar(s, i, c) = '(' then begin
    i := i + 1;
    evalfact := evalexpr(s, i, x);
    if gnbchar(s, i, c) = ')' then
      i := i + 1
    else
      remark('Missing right parenthesis')
  end
  else if c = '[' then
    evalfact := evalcell(s, i, x)
  else if findid(s, i, id) then
    evalfact := evalid(s, i, id, x)
  else
    evalfact := stox(s, i, x);
  if negfact then
    x.frac := -x.frac
end;

```

代碼的新的部份是：

```
...  
else if c = '[' then  
    evalfact := evalcell(s, i, x)  
...  
    1
```

核對左括號指示格子參考；如找到一個 `evalcell` 就被呼叫，
自公式字串中，抄取座標，並回轉格子參考一個值。

```
function evalcell(var s: string; var i: integer; var x: xreal): xresult;  
( Evaluate cell reference )  
  
var  
    cp1: cellptr;  
    col, row: integer;  
  
begin ( evalcell )  
    x := zero;  
    evalcell := OK;  
    getcoord(s, i, col, row);  
    cp1 := findcell(col, row);  
    if cp1 <> NIL then  
        if cp1^.fp <> NIL then  
            with cp1^.fp do begin  
                x := cellval;  
                evalcell := cellstat  
            end  
        end  
    end;  
    1
```

如果參考格子不在表上（自 `findcell` 常式返回一個 `NIL` 指標來做個記號）或格子包括一個標記（`label`），而不是一個公式（由一個 `NIL` 公式——資訊指標註記之），`evalcell` 常式，回復一個 `xreal 0`，作為格子的值，如果該被參考的格子，是個錯誤（溢位、超下限、或除以 0），`evalcell` 就把訊息，傳回 `evalfact` 常式。

`evalcell` 函數，呼叫 `getcoord`，自公式字串中抽取格子參考，並且翻譯成行與列座標。`getcoord` 必須處理絕對參考和相對參考，茲列舉 `getcoord` 程序如下：

```

procedure getcoord(var s: string; var i, col, row: integer);
{ Get column and row numbers from cell reference }

{ NOTE - uses cell pointer cp declared in calc cell procedure }

var
    relcol, relrow: boolean;

begin { getcoord }
    stoc(s, i, col, row, relcol, relrow);
    if relcol then
        col := col + cp^.cellcol;
    if relrow then
        row := row + cp^.cellrow;
end;

```

上面這部份，是處理相對格子和列參考，爲了求得正確的相對參考座標，**getcoord** 必須知道，正在評估的格子之座標，也就是必須知道 **cp . cellcol** 和 **cp . cellrow**。**cp** 的值，已以一個參數傳送給 **calc cell**，同時，**getcoord** 在 **calc cell** 的巢狀結構裡，它可以接受 **cp**，當這個粒子存在的時候，是有些不清和危險。

藉著 **eval expr**，**eval term**，**eval fact** 和 **eval cell** 等常式，把變動的 **cp** 值，當作參數傳送，就必須加一些很好的說明。

getcoord 呼叫 **stoc**，將公式字串的格子參考，翻譯成一對座標：

```

procedure stoc(var s: string; var i, col, row: integer;
               var relcol, relrow: boolean);
{ Extract coordinate pair from string }

var
    c: char;

begin { stoc }
    if gnbchar(s, i, c) = '[' then
        i := i + 1;
    relcol := (gnbchar(s, i, c) in ['+', '-']);
    col := setparam(s, i, -MAXCOLS + 1, MAXCOLS - 1, 0);
    if gnbchar(s, i, c) = ',' then
        i := i + 1;
    relrow := (gnbchar(s, i, c) in ['+', '-']);
    row := setparam(s, i, -MAXROWS + 1, MAXROWS - 1, 0);
    if gnbchar(s, i, c) = ']' then
        i := i + 1;
end;

```

stoc 還被 **getcoord** 之外的常式呼叫，所以它是一個全盤常式，而非包在 **getcoord** 裡的常式。起初它在 **s** 字串的第 **i** 個字元裡找格子參考，它回轉二個值、分別是 **col** 和 **row**。兩布耳參數 **relcol** 和 **relrow** 告訴常式，那被抽取的行或列，是不是相對的。

setparam 常式，被 **stoc** 呼叫，而成就一個普通功能：自字串抽取一個整數參數，檢查之，以確保介於極小和極大之間，如果不介於二者之間，就指派一個缺設值 (default value)。

```
function setparam(var s: string; var i: integer; min, max, default: integer):  
                                                    integer;  
( Extract integer from string, check against range )  
  
var  
    param: integer;  
  
begin ( setparam )  
    if not stoi(s, i, param) then  
        param := default  
    else if (param < min) or (param > max) then  
        param := default;  
    setparam := param  
end;
```

在 **pascalc** 中 **setparam** 尚被其他常式呼叫。

和函數 (SUM Function)

在 **pascalc** 內，**evalid** 和 **calc** 被改變。在 **calc** 裡，**avalid** 被用來在二元樹狀結構中，尋找前面被定義的值。這些被定義的值，在 **pascalc** 裡，都是格子，而且在 **evalcell** 內都深受照顧。因此，**evalid** 在此就有完全不同的責任：評估何時去定義 functions。

pascalc 只有一個函數：**SUM**，它的文法是：

SUM(<arg1>, <arg2>, ... , <argN>)

SUM 接受任意個引數，各引數用逗號分開；它的結果，

就是這些引數的和。每一個引數，可以是一個式子，或一個格子的範圍。如果引數是一個式子，則它的結果，就是式子的值，如果引數是一個格子範圍，它的和就是在範圍內，所有格子的和。一個格子範圍的結構，是以一個冒號，把兩個格子參考分開：

[<col1>, <row1>]: [<col2>, <row2>]

這個範圍，定義一個矩形面積的報表，其左上角是【< col 1>, < row 1>】，而右下角是【< col 2>, < row 2>】，單一系列的和可由設定< row 1> = < row 2> 取得，單一的行，就由設定< col 1> = < col 2> 而得。格子參考可以是絕對，或是相對。如果沒有第二個格子參考，則其範圍，只包括一個單一格子【< col 1>, < row 1>】。

觀察工作狀況，首先考慮 evalid：

```
function evalid(var s: string; var i: integer; id: identifier; var x: xreal);
    xresult;
{ Evaluate identifier as function reference }
var
    c: char;
    arg: string;

{-----}
{ Modules to be inserted here: }
{      findarg      }
{      evalsum      }
{-----}

begin { evalid }
    if id = 'SUM' then
        evalid := evalsum(s, i, x)
    else begin
        remark('Undefined function');
        x := zero;
        evalid := OK;
        if gnbchar(s, i, c) = '(' then begin { skip argument list }
            while findarg(s, i, arg) do
                { nothing };
            if gnbchar(s, i, c) = ')' then
                i := i + 1
            else
                remark('Missing right parenthesis')
            end
        end
    end
end;
```

evalid 付責核對「自 **findid** 所抽的函數名字」是 **SUM**。
若它不是 **SUM**，**evalid** 就認為它企圖使用一個未定義的函數，
如果有這麼個無定義函數，它就給一個記號，並略過該未定義函數的引數列。

當看到了 **SUM**，**evalid** 就呼叫 **evalsum**，以評估該函數：

```
function evalsum(var s: string; var i: integer; var x: xreal): xresult;
( Evaluate sum function )

var
  status: xresult;
  c: char;
  x1: xreal;
  j: integer;

{-----}
( Modules to be inserted here )
(      summatrix      )
{-----}

begin ( evalsum )
  x := zero;
  status := OK;
  if gnbchar(s, i, c) = '(' then begin
    while (status = OK) and ffindarg(s, i, arg) do begin
      j := 1;
      if gnbchar(arg, j, c) = '[' then
        status := summatrix(arg, j, x1)
      else
        status := evalexpr(arg, j, x1);
      if status = OK then
        status := xadd(x, x1, x)
    end;
    if gnbchar(s, i, c) = ')' then
      i := i + 1
    else
      remark('Missing right parenthesis')
    end;
  evalsum := status
end;
```

evalsum 呼叫 **findarg**，由引數列中，抽取連續的引數，
並且逐一評估之。如引數有一個左括號 **<[>**，**evalsum** 就呼
叫 **summatrix**，把引數當作一個格子範圍；否則，它呼叫 **eva-**
lexpr，將引數視為一個式子。

若 **evalid** 和 **evalsum** 二者，皆呼叫 **findarg** 函數；它即
開始在 **s** 字串的第 **(i+1)** 個字串，找一函數引數，而且就把

它當作字串 **ord** 回轉之。若在註明的位置，沒有找到引數，**findarg** 的函數結果是 **FALSE**；否則就是 **TRUE**。它留下 **i** 指向引數的界標，該定界可能是一個逗號，或是一個右括號：

```
function findarg(var s: string; var i: integer; var arg: string): boolean;
< Extract argument from string >

var
  nparen, nbrack: integer;
  done: boolean;
  c: char;

begin ( findarg )
  arg := '';
  if gnbchar(s, i, c) in ['(',')', chr(0)] then
    findarg := FALSE
  else begin
    nparen := 0;
    nbrack := 0;
    done := FALSE;
    repeat
      i := i + 1;
      c := gnbchar(s, i, c);
      if (c in ['(',')']) and (nparen <= 0) and (nbrack <= 0) then
        done := TRUE
      else if c = chr(0) then
        done := TRUE
      else begin
        addchar(arg, c, MAXSTR);
        if c = '(' then
          nparen := nparen + 1
        else if c = ')' then
          nparen := nparen - 1
        else if c = '[' then
          nbrack := nbrack + 1
        else if c = ']' then
          nbrack := nbrack - 1
        end
      end
    until done;
    findarg := TRUE
  end
end;
```

我們不能因為掃描到一個逗號，或一個右括號，就認為已檢視引數完畢，因為引數本身就包括一個逗號，或一個右括號，例如：

SUM([1,1]:[1,10], (1 + (1/3)), [43,38])

findarg 回轉字串 “**【1,1】 :【1,10】**” 做為第一個引數，而

不是把“1”當做第一個引數。同理，第二個引數的兩個右括號，都不是引數的定界 (delimit)。

findarg 是被用來計算，不平衡的左小括號 (parentheses **mparen**) 和左中括號 (brackets) **nbrack**，引數只有遇到逗號或右小括號，而且 **mparen** 和 **nbrack** 都小於或等 0，才停止。

summatrix 程序如下：

```
function summatrix(var s: string; var i: integer; var total: xreal): xresult;
{ Sum up matrix of cells }

var
  status: xresult;
  x1, x2, y1, y2, x: integer;
  cp: cellptr;
  c: char;
  done: boolean;

begin { summatrix }
  total := zero;
  status := OK;
  getcoord(s, i, x1, y1);
  if gnbchar(s, i, c) = ':' then begin
    i := i + 1;
    getcoord(s, i, x2, y2)
  end
  else begin
    x2 := x1;
    y2 := y1
  end;
  if (x1 >= 1) and (x1 <= MAXCOLS) and (y1 >= 1) and (y1 <= MAXROWS) and
    (x2 >= 1) and (x2 <= MAXCOLS) and (y2 >= 1) and (y2 <= MAXROWS)
    then begin
    x := x1;
    while (x <= x2) and (status = OK) do begin
      cp := colptr[x];
      done := FALSE;
      while (cp <> NIL) and (status = OK) and not done do begin
        if cp^.cellrow >= y1 then
          if cp^.cellrow <= y2 then begin
            if cp^.fp <> NIL then
              status := xadd(total, cp^.fp^.cellval, total)
            end
          else
            done := TRUE;
        cp := cp^.downptr
      end;
      x := x + 1
    end
  end;
  summatrix := status
end;
```


這個程序呼叫 **getcoord** 兩次，自引數字串中，取得格子範圍座標。（有時呼叫一次）。其次，再核對整個工作單的範圍，它把這些格子相加，把結果指派給變數 **total**。

重定格式 (Reformatting)

重新計算之後的下一步驟是重定格式，如果一個格子的值已改變，它的顯示欄，必須改變，成新值的被定格式式的字串。當一格的格式改變，就必須重定格式。

表報的重定格式之邏輯與重新計算相似，我們不必顧慮它的次序。

```
procedure reformat;  
{ Reformat sheet }  
  
var  
  r: integer;  
  cp: cellptr;  
  
begin { reformat }  
  center('Reformatting...', statline);  
  for r := 1 to MAXROWS do begin  
    cp := rowptr[r];  
    while cp <> NIL do begin  
      formatcell(cp);  
      cp := cp^.rightptr  
    end  
  end;  
  eraseline(statline)  
end;
```

真正的工作由可以形成單獨 cell 的 **formatcell** 所完成。

第一、**formatcell** 程序決定使用那些「格式化參數」。如果使用人已為格 (**usecolfmt** 是 **FALSE**) 鍵入格式參數，而且參數 (在 **cellfmt** 之內) 被使用。否則，使用人並未註明格的行 (**usesheetfmt** **[cellcol]** 是 **TRUE**)，表報格式參數就用缺設值。否則，就以 **colfmt** **[cellcol]** 說明行格式。

如果格的狀況是 **not ok**，格的顯示欄就設為 **Error**。（只

```

procedure formatcell(cp: cellptr);
{ Format numeric cell display }

var
  s: string;
  f: formatrec;

{-----}
{ Modules to be inserted here: }
{      xtos      }
{      rightjust }
{-----}

begin { formatcell }
  if cp <> NIL then
    with cpn do
      if fp <> NIL then
        with fpn do begin
          if cellstat = OK then begin
            if not usecolfmt then
              f := cellfmt
            else if not usesheetfmt[cellcol] then
              f := colfmt[cellcol]
            else
              f := sheetfmt;
            cellstat := xtos(cellval, f.fmt, f.ndigs, s);
            rightjust(s, f.width)
          end;
          if cellstat <> OK then
            s := 'Error';
          erasestr(display);
          if not storestr(s, display) then
            memout
          end
        end;
      end;
    end;
end;

```

要發生：溢位，超下限，除以零等，就設為 error）。

在 `formatcell` 內同時使用 `xtos` 和 `rightjust` 二常式；而 `xtos` 在第四章出現過了，而 `rightjust` 見第七章。在第四章內，`xtosci` 和 `xtofix` 二常式，都在 `xtos` 的巢狀結構之內。

這完成了鍵入公式和顯示其數值的代碼。而且，這個設計寫了編譯和測試。

命令 (Commands)

最後一個重要目的，是實行表 8-1 的命令表列。該 `doco-`

mmand 常式，在主程式內被呼叫：

```

procedure docommand(ch: char);
( Do user command )

(-----)
( Modules to be inserted here: )
(      getacc      )
(      docopy      )
(      adjustcols  )
(      adjustrows  )
(      doerase     )
(      fnttos      )
(      stofmt      )
(      getformat   )
(      doformat    )
(      doinsert    )
(      doloading   )
(      dodispmem   )
(      doprint     )
(      doquit      )
(      dosave      )
(      dotoggle    )
(-----)

begin { docommand }
  if ch = '/' then begin
    eraseline(inpline);
    posstr('Command? (C/E/F/I/L/M/P/Q/S/T):', 0, inpline);
    ch := getkey(ch, cmdchars, TRUE);
    eraseline(inpline);
    case ch of
      'C': docopy;
      'E': doerase;
      'F': doformat;
      'I': doinsert;
      'L': doloading;
      'M': dodispmem;
      'P': doprint;
      'Q': doquit;
      'S': dosave;
      'T': dotoggle;
    end
  end
  end
  else if ch = '!' then begin
    recalc;
    reformat;
    disp sheet(firstcol, firstrow)
  end
  else if ch = '>' then begin
    posstr('Go to', 0, inpline);
    getacc(6,inpline,1,MAXCOLS,curscol,1,MAXROWS,cursrow,curscol,cursrow);
    eraseline(inpline)
  end;
  setcursor(curscol, cursrow)
end;

```

docommand 的參數 **ch**，可能是個驚歎號< ! >、或是一個大於的符號(>)、或是一個斜線< / >。驚歎號表示重新

計算，重定格式，重新顯示整張表；它可能與呼叫 **recalc**、**reformat** 和 **dispsheet** 同時出現。

「大於」鍵，是被用來搬移一個特定的格，**pascal** 需要使用者，為格鍵入行、列座標，重置全盤變數 **curscol** 和 **currow**，再呼叫 **setcursor** 搬移游標給那格，重定「窗」。

getacc（代表「取得絕對格座標」）被呼叫，以便自使用者那，取得行和列座標。座標的上、下界，和缺設值都被送到常式；座標以參數 **col** 和 **row** 回轉之，**getacc** 設計如下：

```
procedure getacc(crtcol, crtrow, xmin, xmax, xdef, ymin, ymax, ydef: integer;
                 var col, row: integer);
{ Get absolute cell coordinate from user }
begin { getacc }
  posstr('[...]', crtcol, crtrow);
  col := getint(crtcol + 1, crtrow, xmin, xmax, xdef, TRUE);
  row := getint(crtcol + 4, crtrow, ymin, ymax, ydef, TRUE)
end;
```

仔細留意重計算和 **go-to** 命令。如果使用人鍵入斜線，**docommand** 就提示下列十種命令之一：

Command? (C/E/F/I/L/M/P/Q/S/T):

在使用人鍵入這些鍵之後，**docommand** 呼叫對應的常式去執行命令。

簡單命令 (Easy Commands)

當設計大量的常式，而且每一常式，又似乎一樣的重要，最好的法則是「先做容易的」，第一個要寫的是 **doquit** 常式：

```
procedure doquit;  
{ Set up for exit }  
  
begin { doquit }  
  alldone := ask('Quit. Are you sure?(Y/N):', inpline, FALSE, TRUE);  
  eraseline(inpline)  
end;
```

要記住，當 **alldone** 是 **TRUE** 時，**Pascalc** 的主要控制環（**loop**）就出口。

dodispmem 常式也非常容易，它必須掃描樹狀表列，把自由塊的大小全部加起來，再顯示其和：

```
procedure dodispmem;  
{ Display free memory }  
  
var  
  p: headerptr;  
  sum: integer;  
  s: string;  
  
begin { dodispmem }  
  sum := 0;  
  p := freelist;  
  repeat  
    sum := sum + HDRSIZE * p^.size;  
    p := p^.next  
  until p = freelist;  
  itos(sum, 0, s);  
  remark(concat(s, ' bytes left'))  
end;
```

dotoggle 相當簡單，它允許使用人去顯示和選擇改變(1)重新計算的次序，和(2)是否自動重算或人工操縱之，茲列舉 **dotoggle**：

```
procedure dotoggle;  
{ Display and (optionally) change recalculation parameters }  
  
var  
  ch: char;  
  s: string;
```

```
begin ( dotoggle )
  posstr('Recalculation: Order or Mode?(O/M):', 0, inpline);
  case getkey(ch, ['O', 'M'], TRUE) of
    'O': begin
      if colcalc then
        center('Recalculation is by column', statline)
      else
        center('Recalculation is by row', statline);
      if ask('Change it?(Y/N):', inpline, FALSE, TRUE) then
        colcalc := not colcalc
      end;
    'M': begin
      if autocalc then
        center('Recalculation is automatic', statline)
      else
        center('Recalculation is manual', statline);
      if ask('Change it?(Y/N):', inpline, FALSE, TRUE) then
        autocalc := not autocalc
      end;
    end;
  end;
  eraseline(inpline);
  eraseline(statline)
end;
```

顯示和改變格式的參數 (Display and Changing Formatting Parameters)

格式命令 **<IF>** 是用來顯示，和改變表報格式參數的選擇（改變流程的行或列），鍵入該命令，則使用者看到提示如下：

Format: Sheet, Column, or One cell?(S/C/O):

使用人按下 **<S>**，以便看表報格式；**<C>** 是看行格式，而 **<O>** 是格的格式。本程式顯示這些格式的需求，詢問使用人，是否要改變它，如使用人希望改變格式，本程式即因應要求而接受新的格式。

doformat 常式如下：

```
procedure doformat;
( Display and (optionally) change cell format )

var
  s: string;
```

```

ch: char;
f: formatrec;

begin ( doformat )
  posstr('Format: Sheet, Column, or One cell?(S/C/O):', 0, inpline);
  ch := getkey(ch, ['S', 'C', 'O'], TRUE);
  eraseline(inpline);
  case ch of
    'S': begin
      fttos(sheetfmt, s);
      center(concat('Current sheet format is ', s), statline);
      if ask('Change it?(Y/N):', inpline, FALSE, TRUE) then begin
        getformat(sheetfmt);
        setcolpos;
        labelcols(firstcol);
        reformat;
        disp-sheet(firstcol, firstrow)
      end
    end;
    'C': begin
      if usesheetfmt[curscol] then
        f := sheetfmt
      else
        f := colfmt[curscol];
      fttos(f, s);
      center(concat('Current column format is ', s), statline);
      if ask('Change it?(Y/N):', inpline, FALSE, TRUE) then begin
        getformat(f);
        colfmt[curscol] := f;
        usesheetfmt[curscol] := FALSE;
        setcolpos;
        labelcols(firstcol);
        reformat;
        disp-sheet(firstcol, firstrow)
      end
    end;
    'O':
      if curcp <> NIL then
        with curcp^ do
          if fp <> NIL then
            with fp^ do begin
              if not usecolfmt then
                f := cellfmt
              else if usesheetfmt[cellcol] then
                f := sheetfmt
              else
                f := colfmt[cellcol];
              fttos(f, s);
              center(concat('Current cell format is ', s), statline);
              if ask('Change it?(Y/N):', inpline, FALSE, TRUE) then begin
                getformat(f);
                cellfmt := f;
                usecolfmt := FALSE;
                formatcell(curcp);
                disprow(cellrow)
              end
            end
          end
        end
      end
  end
end;
end;

```

如果，報表格式或一行格式被改變，報表就被重定，重新顯示；**setcolpos** 常式，就被呼叫，重新設定 **colpos** 列陣。

使用 **getformat** 即得到格式參數：

```
procedure getformat(var fmtrec: formatrec);
( Get new format )

var
  i: integer;
  s: string;
begin ( getformat )
  fmttos(fmtrec, s);
  center('New format?: _____', inpline);
  getstring(s, 6, (MAXCRTCOL + 7) div 2, inpline, s,
    ['0'..'9', fmtchar[FIXEDPOINT], fmtchar[SCIENTIFIC], '.', ''], TRUE);
  eraseline(inpline);
  i := 1;
  stofmt(s, i, fmtrec)
end;
```

格式的參數像字串一樣被鍵入，（方法與其他 **get** 常式相同）；我們必須有一個方法，以字串來代表其三個格式的參數，並且在字串與參數之間翻譯。

我們使用一個，像在 FORTRAN 中常用的轉換方法；第一個字元是一個「說明模式的模式」的字母：<F> 表 **FIXEDPOINT**，<S> 表 **SCIENTIFIC**。其次是寬度欄，十進位小數點，再接著是數位的個數。例如，**initpc** 常式以下列代碼來定表報格式：

```
...
sheetfmt.fmt := FIXEDPOINT;
sheetfmt.width := 10;
sheetfmt.ndigs := 2;
...
```

這些參數可由字串 <F10.2> 代表。<S0.5> 代表科學格式，在小數點後面有五位數，寬度為 0（該數字字串向左調整。）

• 8 電子工作表格 •

stofmt 程序，把一個字串，翻譯成一個格式的參數記

錄：

```
procedure stofmt(var s: string; var i: integer; var fmtrec: formatrec);
{ Extract format parameters from string }

var
  c: char;

begin { stofmt }
  with fmtrec do begin
    if gchar(s, i, c) = fmtchar[FIXEDPOINT] then begin
      fmt := FIXEDPOINT;
      i := i + 1
    end
    else if c = fmtchar[SCIENTIFIC] then begin
      fmt := SCIENTIFIC;
      i := i + 1
    end;
    width := setparam(s, i, MINWIDTH, MAXWIDTH, width);
    if gnbchar(s, i, c) = '.' then begin
      i := i + 1;
      ndigs := setparam(s, i, MINDIGS, MAXDIGS, ndigs)
    end
  end
end;
end;
```

fmttos 常式執行相反的函數，把一個格式的參數集合翻

譯成一字串：

```
procedure fmttos(fmtrec: formatrec; var fmtstr: string);
{ Convert format parameters into Fortran-like format specification }

var
  s1: string;

begin { fmttos }
  fmtstr := '';
  with fmtrec do begin
    addchar(fmtstr, fmtchar[fmt], MAXSTR);
    itos(width, 0, s1);
    fmtstr := concat(fmtstr, s1, '.');
    itos(ndigs, 0, s1);
    fmtstr := concat(fmtstr, s1)
  end;
end;
end;
```

抄的命令 (Copy command)

Copying 可能是 Pascal 中，最有用的一個命令。總之，這個命令，是抄寫一個或多個特定的「格的範圍」，你可以抄標記，公式格，或二者都抄。

當使用本命令常式，程式提示：

```
Copy from [...,...]:[...,...] to [...,...]:[...,...] step [...,...]
```

程式要求使用人，首先鍵入格的原始範圍，再鍵入格的目的範圍。

範圍可能註明表報的任意矩形部份。每一格，單一行、單一行，都是該矩形的特殊案件，下列以「矩陣」來討論一般矩形部份。

因為任意範圍，可以明定一格、一行、一列、或一矩陣，我們總共有16種不同狀況待抄，茲舉要討論如下：

1 自一格抄到另一格：

原始和目的範圍都是單格，“step”就不需要了，例如

```
Copy from [1,15]:[1,15] to [10,4]:[10,4] step [...,...]
```

將【1,15】處抄到【10,4】。

2 抄一行到一格：

根據鍵入行的頂和底座標，來說明原始範圍。而目的範圍就當做是單格鍵入，行的頂就被抄了，行的其餘部份就被抄到格的頂點之下，step也不需要了，例如：

```
Copy from [2,10]:[2,15] to [5,5]:[5,5] step [...,...]
```

自【2,10】到【2,15】六格，抄到一行起自【5,5】到【5,10】去。

3. 抄一列到一格：

與案例 2 相似，只是以列取代行，使用人只要說明要抄之列的左、右座標。目的範圍是那被抄的最左列；其餘的，就抄到上述的右方，step 也不需要。

4. 抄一個矩陣到一格：

這是前三種的一般性案件，不是一個單一格，一個單行、一個單列，而是原始的完整的範圍。並不需要 step 值，例如：

Copy from [1,1]:[10,10] to [21,21]:[21,21] step [...,...]

自矩陣左上角【1,1】與右下角【10,10】的矩陣，抄 100 格（10 × 10 矩陣）到相同——大小矩陣，其左上角是【21,21】。

5. 抄一格到一行：

與案件一單一格有相同的原始範圍，目的範圍被鍵入，做為行的頂與底座標，而這二座標，是使用人要抄單一格的目的地。爲了要控制被抄的格所在之列，或每一其他列，每一第三列等，需要 step 大小值。例如，

Copy from [5,10]:[5,10] to [10,1]:[10,10] step [...,1]

在【5,10】的格，抄到 10 格【10,1】，【10,2】，……，【10,10】，如果 step 的大小是 2，而不是 1，則格必然是被抄到每一其他的列：【10,1】，【10,3】……【10,9】。

6. 抄一格到一列

與第五種情況相似，使用人註明一個單格，作為一個原始和左列座標、右列座標，做為目的。需要控制抄的空間，所以必須要求行 step 大小。

7. 抄一格到一矩陣：

這是情五、六的一般性。原始是一單格，而使用人必須註明目的範圍。無論是行 step 大小或到 step 的大小，都必須被要求。

8. 抄一行到一列：

必須申明原始範圍是一行，而目的是一列。這是情況 2 的一般性狀況；並不註明自原始行抄一單行，而產生行的一份拷貝，使用人說明起點的一水平列，做為被抄的行，而產生拷貝一行以上。目的列的大小，是需要的，該值是被用來，說明抄行的不同空間。

9. 抄一列到一行：

與情況八類似，而且是情況 3 的一般性：被抄的個數是由垂直行的原始列組成，程式需要 step 大小，做為控制行的空間。

其他的情況（諸如：行到行、行到矩陣、列到列、列到矩陣、矩陣到行、矩陣到列，和矩陣到矩陣）是這九種案例的一般性。坦白地說，這些案例被用來做粒子的應用，是難以理解的，然而使用人也許發現這伸縮性，對有些事物是有好處的。這個程式特性做些人為的限制，是錯誤的，甚且我們也無法確實理解，為什麼會利用這些特性。docopy 的虛擬代碼像：

• 8 電子工作表格 •

```

begin
    get source range
    get destination range
    get destination step sizes, if necessary
    for each cell in destination range
        copy source range
    if we ran out of memory
        inform the user
    if recalculation is automatic
        recalculate and reformat sheet
    redisplay sheet
end

```

改寫成 PASCAL:

```

procedure docopy;
{ Do copy command }

var
    x1, x2, x3, x4, y1, y2, y3, y4, xstep, ystep, xdest, ydest: integer;
    success: boolean;

{-----}
{ Modules to be inserted here: }
{      copymatrix      }
{-----}

begin { docopy }
    posstr('Copy from [...]:[...] to [...]:[...] step [...]',
           0, inpline);
    getacc(10, inpline, 1, MAXCOLS, curscol, 1, MAXROWS, cursrow, x1, y1);
    getacc(19, inpline, x1, MAXCOLS, x1, y1, MAXROWS, y1, x2, y2);
    getacc(31, inpline, 1, MAXCOLS, x1, 1, MAXROWS, y1, x3, y3);
    getacc(40, inpline, x3, MAXCOLS, x3, y3, MAXROWS, y3, x4, y4);
    xstep := 1;
    ystep := 1;
    if x4 > x3 then
        xstep := getint(55, inpline, 1, MAXCOLS, xstep, TRUE);
    if y4 > y3 then
        ystep := getint(58, inpline, 1, MAXROWS, ystep, TRUE);
    center('Copying...', statline);
    success := TRUE;
    xdest := x3;
    while (xdest <= x4) and success do begin
        ydest := y3;
        while (ydest <= y4) and success do begin
            success := copymatrix(x1, y1, x2, y2, xdest, ydest);
            ydest := ydest + ystep;
        end;
        xdest := xdest + xstep;
    end;
    eraseline(statline);
    if not success then
        memout;
    if autocalc then begin
        recalc;
        reformat;
    end;
    disp-sheet(firstcol, firstrow);
    eraseline(inpline);
end;

```

docopy 程序呼叫 **getacc** 四次，以取得要拷貝的源和目的範圍；源範圍是 **【x1,y1】：【x2,y2】**，而目的範圍是 **【x3,y3】：【x4,y4】**。

docopy 程序呼叫 **copymatrix** 去抄源範圍到目的範圍的一格，抄的時候，如果沒有記憶位置供程式運轉，則 **copymatrix** 回轉 **FALSE**，立刻使抄的過程終止。

copymatrix 自源範圍，把每一個非空格、抄到特定目的。

```
function copymatrix(x1, y1, x2, y2, xdest, ydest: integer): boolean;
{ Copy pascal matrix to destination }

var
  x, y: integer;
  cp: cellptr;
  success, done: boolean;

{-----}
{ Modules to be inserted here: }
{       copycell       }
{-----}

begin { copymatrix }
  x := x1;
  success := TRUE;
  while (x <= x2) and success do begin
    cp := colptr[x];
    done := FALSE;
    while (cp <> NIL) and (not done) and success do begin
      y := cp^.cellrow;
      if y >= y1 then
        if y <= y2 then
          success := copycell(cp, xdest + (x - x1), ydest + (y - y1))
        else
          done := TRUE;
      cp := cp^.downptr
    end;
    x := x + 1
  end;
  copymatrix := success
end;
```

copymatrix 用行來抄源範圍；只有在特殊的理由之下，需要指定抄二個或二個以上的源元素、到相同目的地的格上，

才要選擇抄的次序。

用 **copycell** 將非一空格抄到其他位置：

```
function copycell(cp: cellptr; col, row: integer): boolean;
{ Copy cell to specified position }

var
  newcp: cellptr;
  success: boolean;

begin { copycell }
  success := TRUE;
  if (col >= 1) and (col <= MAXCOLS) and
    (row >= 1) and (row <= MAXROWS) then begin
    newcp := findcell(col, row);
    if newcp <> NIL then
      erasercell(newcp);
    success := storecell(col, row, newcp);
    if success then
      with newcp do begin
        if cp^.display <> NIL then
          success := storestr(cp^.display, display);
        if cp^.fp <> NIL then begin
          success := storeform(fp);
          if success then
            with fp do begin
              cellstat := cp^.fp^.cellstat;
              cellval := cp^.fp^.cellval;
              usecolfmt := cp^.fp^.usecolfmt;
              cellfmt := cp^.fp^.cellfmt;
              if cp^.fp^.formula <> NIL then
                success := storestr(cp^.fp^.formula, formula)
            end
          end
        end
      end;
    success := success;
  end;
end;
```

本常式首先檢查格座標，並確定其在表報限制下被抄。如果一格流程位置被抄寫，它就被抹除。新格由 **storecell** 建立，源格的內容，以取用記憶——配置程序來抄到目的格，如抄寫成功，就像 **copymatrix**，**copycell** 一樣，回轉 **TRUE**，否則回轉 **FALSE**。

抹除的命令 (Erase Command)

Pascalc 及時抹除電子的表報資訊，這工作由命令 **</E>** 完成之。當給了這個命令，程式就提示：

Erase: One cell, Columns, Rows, or Sheet?(O/C/R/S):

使用人按 **<O>** 鍵，抹除游標位置上的格，**<C>** 鍵抹除若干行，**<R>** 抹除若干列，**<S>** 抹除表報上所有的資訊，若一格被抹除，**pascalc** 詢問。

Erase cell. Are you sure?(Y/N):

如使用人鍵入 **<Y>**，該格被抹除；否則就單獨留下。抹除行，**Pascalc** 詢問

Erase how many columns?:

抹除行數依使用人鍵入資料決定之，鍵入 0 捨棄表報，所謂抹除行，包括游標及游標右方的位置。該命令下達後，游標右方所有的行都被移走。

列消除與行消除相似，**Pascalc** 詢問有多少列待消除，鍵入 0 值，表報就失效了，所謂抹除列，包括游標流程和它下面的列全部消除之，未抹除的列就向上搬移。

列消除和行消除可以互相變換的設計，比較安全，却會增加幾許困難。（讀者可以兩種都試驗看看，選擇自己比較喜歡的）。

如要抹除整張表報，使用人會被詢問二次以便確定首先得到提示：

Erase sheet. Are you sure?(Y/N):

然後是：

Erase sheet. ARE YOU CERTAIN?(Y/N):

這種要求再保證，是避免意外抹除一張表報。

doerase 常式處理格和表報的抹除，如要複雜一點、要求抹除一定要正確，就需仰賴 **erasecols** 和 **eraserows** 了。

doerase 程序如下：

```
procedure doerase;
{ Do erase command }

var
  ch: char;
  c: integer;

{-----}
{ Modules to be inserted here: }
{      erasecols      }
{      eraserows     }
{-----}

begin { doerase }
  posstr('Erase: One cell, Columns, Rows, or Sheet?(O/C/R/S):', 0, inpline);
  ch := getkey(ch, ['O', 'C', 'R', 'S'], TRUE);
  eraseline(inpline);
  case ch of
    'O': begin
      if ask('Erase cell. Are you sure?(Y/N):', inpline, FALSE, TRUE) then begin
        erasecell(curcp);
        if autocalc then begin
          recalc;
          reformat;
          dispsheet(firstcol, firstrow)
        end
      else
        disprow(cursrow)
      end;
      eraseline(inpline)
    end;
    'C':
      erasecols;
    'R':
      eraserows;
    'S': begin
      if ask('Erase sheet. Are you sure?(Y/N):', inpline, FALSE, TRUE) then
        if ask('Erase sheet. ARE YOU CERTAIN?(Y/N):', inpline, FALSE, TRUE) then
          center('Erasing...', inpline);
          for c := 1 to MAXCOLS do
            while colptr[c] <> NIL do
              erasecell(colptr[c]);
            dispsheet(firstcol, firstrow)
          end;
          eraseline(inpline)
        end
      end
    end
  end;
end;
```

再考慮是否確定要抹除行

```
procedure erasecols;  
( Delete columns from sheet )  
  
var  
  n, c: integer;  
  cp: cellptr;  
  
begin ( erasecols )  
  posstr('Erase how many columns?:', 0, inline);  
  n := getint(24, inline, 0, MAXCOLS - curscol + 1, 0, TRUE);  
  if n > 0 then begin  
    center('Erasing...', statline);  
    for c := curscol to curscol + n - 1 do  
      while colptr[c] <> NIL do  
        erasecell(colptr[c]);  
    for c := curscol to MAXCOLS - n do begin  
      colptr[c] := colptr[c + n];  
      usesheetfmt[c] := usesheetfmt[c + n];  
      colfmt[c] := colfmt[c + n];  
      cp := colptr[c];  
      while cp <> NIL do begin  
        cp^.cellcol := c;  
        cp := cp^.downptr  
      end  
    end;  
    for c := MAXCOLS - n + 1 to MAXCOLS do begin  
      colptr[c] := NIL;  
      usesheetfmt[c] := TRUE  
    end;  
    setcolpos;  
    adjustcols(n);  
    eraseline(statline);  
    if autocalc then begin  
      recalc;  
      reformat  
    end;  
    labelcols(firstcol);  
    dispheet(firstcol, firstrow)  
  end;  
  eraseline(inline)  
end;
```

對於每一次抹除行，**erasecols** 重複呼叫 **erasecell** 消除行的頂格、直到行為空才停止。

erasecols 移走所有的行。當消除了 **n** 行之後，它右方的 **n** 行，全向左移。這可以減化，只要重定行指標，和其餘行的資訊變數。我們還必須重新調整格的 **cellcol** 欄，抹除 **n** 行，就在表報的右方，留下 **n** 個空行，行的指標設定為 **NIL**。

erasecols 尚有在格的公式中調整格參考的任務，以反映

新行引數，這由 **adjustcols** 程序負責；它調整表報的所有公式格，改變他們的格的參考，必要時參考前面相同的格。

調整之後的邏輯非常難，例如，只要在表報上消除 **n** 行，則考慮公式格包含一參考 [**col,row**] ；行參考 **col** 可以是絕對或相對。

有三種不同的案例需要改變格參考：

- 1 行參考是絕對的，它所指的行必須搬移，只要以 [**col-n** , **row**] 取代 [**col** , **row**]。
- 2 行參考是相對的，如公式格和參考格已被搬移，彼此相對參考，就只須調整，亦即與被消除之行，方向相反。如公式格，在被消除列的左方，而參考格在被消除列的右方，則參考以 [**col-n** , **row**]，取代 [**col** , **row**]。
- 3 公式格在被消除行的右方，參考格在左方，此時 **col** 是負數、其值小於負 **n**。（因格子彼此搬移 **n** 行。於是，參考以 [**col + n** , **row**] 取代 [**col** , **row**]。（最後一種案件，最難理解；事實上 **Pascal** 的早期版本處理錯誤）

adjustcols 程序如下：

```
procedure adjustcols(n: integer);
( Adjust column references in sheet formulas )

var
  c, col, row, i: integer;
  cp: cellptr;
  s, s1: string;
  change, relcol, relrow: boolean;
  ch: char;

begin ( adjustcols )
  for c := 1 to MAXCOLS do begin
    cp := colptr[c];
    while cp <> NIL do begin
      with cp^ do
        if fp <> NIL then
          with fp^ do
            if formula <> NIL then begin
              s := '';
              i := 1;
              change := FALSE;
              while gnbchar(formula^, i, ch) <> chr(0) do
                if ch <> '[' then begin
                  addchar(s, ch, MAXSTR);
                  i := i + 1;
                end;
            end;
          end;
      cp := cp^.next;
    end;
  end;
```

• 高等 Pascal 程式設計技巧 •

```

end
else begin
  stoc(formula^, i, col, row, relcol, relrow);
  if relcol then begin
    if (c >= curscol) and
      (c + n + col < curscol) then begin ( Case 3 )
      change := TRUE;
      col := col + n
    end
    else if (c < curscol) and
      (c + col >= curscol) then begin ( Case 2 )
      change := TRUE;
      col := col - n
    end
    else if (col >= curscol) then begin ( Case 1 )
      col := col - n;
      change := TRUE
    end;
    ctos(col, row, relcol, relrow, s1);
    s := concat(s, s1)
  end;
  if change then begin
    erasestr(formula);
    if not storestr(s, formula) then
      memout
    end
  end;
  cp := cp^.downptr
end
end
end;

```

adjustcols 根據需要，在表報上找公式，改變行參考。
若一公式被改變，舊版就被 **erasestr** 消除了，而新版就由 **storestr** 儲存之。

eraserows 程序與 **erasecols** 程序相似；由於不必考慮「行的格式」，**eraserow** 非常簡單，茲列舉如下：

```

procedure eraserows;
( delete rows from sheet )

var
  n, r: integer;
  cp: cellptr;

begin ( eraserows )
  posstr('Erase how many rows?:', 0, inpline);
  n := getint(21, inpline, 0, MAXROWS - cursrow + 1, 0, TRUE);
  if n > 0 then begin
    center('Erasing...', statline);
    for r := cursrow to cursrow + n - 1 do
      while rowptr[r] <> NIL do
        erase cell(rowptr[r]);
    for r := cursrow to MAXROWS - n do begin
      rowptr[r] := rowptr[r + n];
      cp := rowptr[r];
      while cp <> NIL do begin

```

• 8 電子工作表格 •

```

        cp^.cellrow := r;
        cp := cp^.rightptr
    end
end;
for r := MAXROWS - n + 1 to MAXROWS do
    rowptr[r] := NIL;
adjustrows(n);
eraseline(statline);
if autocalc then begin
    recalc;
    reformat
end;
dispsheet(firstcol, firstrow)
end;
eraseline(inpline)
end;

```

同時需要 adjustrows 程序，這與 adjustcol 類似：

```

procedure adjustrows(n: integer);
( Adjust row references in sheet formulas )

var
    r, col, row, i: integer;
    cp: cellptr;
    s, s1: string;
    change, relcol, relrow: boolean;
    ch: char;

begin ( adjustrows )
    for r := 1 to MAXROWS do begin
        cp := rowptr[r];
        while cp <> NIL do begin
            with cp do
                if fp <> NIL then
                    with fp do
                        if formula <> NIL then begin
                            s := '';
                            i := 1;
                            change := FALSE;
                            while gnbchar(formula^, i, ch) <> chr(0) do
                                if ch <> '[' then begin
                                    addchar(s, ch, MAXSTR);
                                    i := i + 1
                                end
                                else begin
                                    stoc(formula^, i, col, row, relcol, relrow);
                                    if relrow then begin
                                        if (r >= cursrow) and
                                           (r + n + row < cursrow) then begin ( Case 3 )
                                            change := TRUE;
                                            row := row + n
                                        end
                                        else if (r < cursrow) and
                                           (r + row >= cursrow) then begin ( Case 2 )
                                            change := TRUE;
                                            row := row - n
                                        end
                                    end
                                else if (row >= cursrow) then begin ( Case 1 )
                                    row := row - n;
                                    change := TRUE
                                end;
                                    ctos(col, row, relcol, relrow, s1);

```

• 高等Pascal 程式設計技巧 •

```

      s := concat(s, s1)
    end;
  if change then begin
    erasestr(formula);
    if not storestr(s, formula) then
      memout
    end;
  end;
  cp := cp^.rightptr
end
end;
end;

```

我們必須考慮，當一格公式參考正待消除之格時，adjustcols 和 adjustrows 會發生什麼？

插入命令 (Insert Command)

插入命令 (</I>) 用來插入若干空白列或行，移走若干列或行，增加空間。這與剪貼表格類似。

鍵入命令後，Pascalc 問

Insert. Columns or Rows?(C/R):

<C> 表插入行，<R> 插入列。

doinstert 程序如下：

```

procedure doinsert;
{ Insert empty rows or columns in sheet }

var
  ch: char;

{-----}
{ Modules to be inserted here: }
{      insertcols      }
{      insertrows      }
{-----}

begin { doinsert }
  posstr('Insert. Columns or Rows?(R/C):', 0, inpline);
  ch := getkey(ch, ['C', 'R'], TRUE);
  eraseline(inpline);
  case ch of
    'R':
      insertrows;
    'C':
      insertcols;
  end
end;

```

doinsert 呼叫 **insertcol** 插入行，呼叫 **insertrows** 插入列。

insertcol 如下：

```

procedure insertcols;
( Insert columns )

var
  c, n: integer;
  cp: cellptr;
  ok: boolean;

begin ( insertcols )
  posstr('Insert how many columns?:', 0, inpline);
  n := getint(25, inpline, 0, MAXCOLS - curscol + 1, 0, TRUE);
  if n > 0 then begin
    ok := TRUE;
    c := MAXCOLS;
    while (c >= MAXCOLS - n + 1) and ok do begin
      ok := (colptr[c] = NIL);
      c := c - 1;
    end;
    if not ok then begin
      center('Warning: insertion will erase non-empty cells.', statline);
      ok := ask('Insert anyway?(Y/N):', inpline, FALSE, TRUE);
      eraseline(statline);
    end;
    if ok then begin
      center('Inserting...', statline);
      for c := MAXCOLS downto MAXCOLS - n + 1 do
        while colptr[c] <> NIL do
          erasecell(colptr[c]);
      for c := MAXCOLS downto curscol + n do begin
        colptr[c] := colptr[c - n];
        usesheetfmt[c] := usesheetfmt[c - n];
        colfmt[c] := colfmt[c - n];
        cp := colptr[c];
        while cp <> NIL do
          cp^.cellcol := c;
          cp := cp^.downptr;
        end;
      end;
      for c := curscol + n - 1 downto curscol do begin
        colptr[c] := NIL;
        usesheetfmt[c] := TRUE;
      end;
      setcolpos;
      adjustcols(-n);
      eraseline(statline);
      if autocalc then begin
        recalc;
        reformat;
      end;
      labelcols(firstcol);
      dispheet(firstcol, firstrow);
    end;
  end;
  eraseline(inpline);
end;

```

insertcols 首先詢問使用人，要插入多少空白行； ϕ 表示該表報絲毫不變，接著檢查插入的行，是否與非——空格（右方的 n 行）是否相衝突，如相衝突，它要求使用人繼續之前確定一下。

其次，**insertcols** 消除表報的右方 n 行，空出空間給新行。若這些行中沒有非——空行，所有資訊都會喪失。再者，游標位置和表報右邊之間所有的行，由「重新指派行指標和相關變數和與 **erasescols** 內的 **cellcol** 欄一樣重新置定」，向右搬移 n 行。最後，插入的新行之指標，被設定為 **NIL**。

在插入 n 個空白行之後，**insertcols** 呼叫 **adjustcols** 去調整公式內的行參考（就像 **erasescols** 一樣）。在這種案例中，全部的調整，都是由行消除的要求而保留下來的；**adjustcols** 引數是 $-n$ ，而非 n 。

insertrows 與 **insertcols** 相似：

```
procedure insertrows;
( Insert rows )

var
  r, n: integer;
  cp: cellptr;
  ok: boolean;

begin ( insertrows )
  posstr('Insert how many rows?:', 0, inline);
  n := getint(22, inline, 0, MAXROWS - cursrow + 1, 0, TRUE);
  if n > 0 then begin
    ok := TRUE;
    r := MAXROWS;
    while (r >= MAXROWS - n + 1) and ok do begin
      ok := (rowptr[r] = NIL);
      r := r - 1;
    end;
    if not ok then begin
      center('Warning: insertion will erase non-empty cells.', statline);
      ok := ask('Insert anyway?(Y/N):', inline, FALSE, TRUE);
      eraseline(statline);
    end;
    if ok then begin
      center('Inserting...', statline);
      for r := MAXROWS downto MAXROWS - n + 1 do
        while rowptr[r] <> NIL do
          erasecell(rowptr[r]);
      for r := MAXROWS downto cursrow + n do begin
        rowptr[r] := rowptr[r - n];
        cp := rowptr[r];
        while cp <> NIL do begin
```



```

        cp^.cellrow := r;
        cp := cp^.rightptr
    end
end;
for r := cursrow + n - 1 downto cursrow do
    rowptr[r] := NIL;
adjustrows(-n);
eraseline(statline);
if autocalc then begin
    recalc;
    reformat
end;
dispsheet(firstcol, firstrow)
end
end;
eraseline(inpline)
end;

```

載入與儲存 (Loading and saving)

爲了充分使用，Pascalc 必須能把在記憶器內的表報存到磁碟，下次也能重新回到記憶內。這就需要儲存命令（</S>）和載入命令（</L>）；儲存命令必須把流程表報方面足夠的資訊，寫到磁碟上，俾使其重新載入，回到記憶體時，其所得到的表報是與當初儲回去的同一表報。所以，磁碟檔需要一些儲存體，儲存有關表報的資訊——不僅是包含在 cells 內的資訊。而且還有重新計算模式，和格式。

我們以本文檔 (text file) 的形式，將表報存進磁碟。這個檔案可以用編輯器來檢查和更改，在大部份的系統內，都適用。這個決定，也允許我們，使用第 5 章的分開編譯的 text-stuff 單元，及其引導 pascalc 在主程式使用 uses 敘述。

dosave 程序首先詢問用什麼檔案名來存表報。所存表報包括三部份：(1)、儲存表報的參數。(2)、存行的格式、和(3)、表格中每一個非——空格的資訊。每一步驟都是由個別的模式實行：writesp，writecols，和writecell等。如果有錯誤，這三個常式就把輸出／入的錯誤信號狀態傳回 dosave，使用人才可能確實掌握執行狀況。dosave 常式如下：

• 高等 Pascal 程式設計技巧 •

```

procedure dosave;
{ Save sheet to disk }

var
  f: tfile;
  tf: tfrec;
  status: iostatus;
  name: filename;
  c: integer;
  cp: cellptr;

{-----}
{ Modules to be inserted here: }
{      writesp      }
{      writecols    }
{      writecell    }
{-----}

begin { dosave }
  posstr('Save sheet. Filename?:', 0, inpline);
  gettfname(name, 22, inpline, '');
  if name <> '' then begin
    center(concat('Saving to ', name), statline);
    if tfcreate(f, tf, name, status) = GOODIO then begin
      if writesp(f, tf, status) = GOODIO then
        if writecols(f, tf, status) = GOODIO then begin
          c := 1;
          while (c <= MAXCOLS) and (status = GOODIO) do begin
            cp := colptr[c];
            while (cp <> NIL) and (status = GOODIO) do begin
              status := writecell(f, tf, cp, status);
              cp := cp^.downptr
            end;
            c := c + 1
          end;
          if status <> GOODIO then
            remark('Error writing cell data')
        end
      else
        remark('Error writing column data')
    else
      remark('Error writing sheet parameters');
    if tfclose(f, tf, status) <> GOODIO then
      remark('Error closing file')
  end
  else
    remark(concat('Can't create ', name));
  eraseline(statline)
end;
  eraseline(inpline)
end;

```

doload 常式與 **dosave** 常式平行，輸入檔案名字是因應使用人的要求而賦予的，其次諸如表報參數，行參數，格資料都是由分隔的常式 **readsp**、**readcols**，和 **readcell** 讀入的。

```

procedure doload;
{ Load sheet from disk }

var

```

• 8 電子工作表格 •

```
f: tfile;
tf: tfrec;
status: iostatus;
name: filename;

{-----}
{ Modules to be inserted here: }
{      readsp      }
{      readcols    }
{      readcell    }
{-----}

begin ( doloop )
  posstr('Load sheet. Filename?:', 0, inpline);
  gettfname(name, 22, inpline, '');
  if name <> '' then begin
    center(concat('Loading ', name), statline);
    if tfopen(f, tf, name, status) = GOODIO then begin
      if readsp(f, tf, status) = GOODIO then
        if readcols(f, tf, status) = GOODIO then begin
          while readcell(f, tf, status) = GOODIO do
            ( nothing );
          if status <> ENDFILE then
            remark('Error reading cell data')
          end
        else
          remark('Error reading column data')
        else
          remark('Error reading sheet parameters');
      if tfclose(f, tf, status) <> GOODIO then
        remark('Error closing file')
      end
    else
      remark(concat('Can't open ', name));
    eraseline(statline);
    setcolpos;
    labelcols(firstcol);
    disp sheet(firstcol, firstrow)
  end;
  eraseline(inpline)
end;
```

讀和寫的表報參數，是最容易的部分，首先考慮 writesp：

```
function writesp(var f: tfile; var tf: tfrec; var status: iostatus): iostatus;
{ Write sheet parameters to disk }

var
  s, s1: string;

begin ( writesp )
  s := '';
  addchar(s, boochar[autocalc], MAXSTR);
  addchar(s, boochar[colcalc], MAXSTR);
  fttos(sheetfmt, s1);
  s := concat(s, s1);
  addchar(s, chr(13), MAXSTR);
  writesp := tfwrite(f, tf, s, status)
end;
```

這常式將參數 **autocalc** , **colcalc** 和 **sheetfmt** 翻譯成一個字串，並將字串當作本文檔的第一行：布耳 **TRUE** 轉換為一個 **<Y>** 字元，布耳 **FALSE** 則為 **<N>** 字元。於是乎，如 **autocalc** 是 **TRUE** , **colcalc** 是 **FALSE** 而 **sheetfmt** 是 **<F 10.2>**，本文檔的第一行變成

YNF 10.2

readsp 常式讀到該檔的第一行，就翻譯回去，成為表報參數 **autocalc** , **colcalc** , 和 **sheetfmt** 等，而 **readsp** 如下：

```
function readsp(var f: tfile; var tf: tfrec; var status: iostatus): iostatus;
( Read sheet paramters from file )

var
  s: string;
  i: integer;
  ch: char;

begin ( readsp )
  if tftread(f, tf, s, MAXSTR, status) = GOODIO then begin
    autocalc := (gchar(s, 1, ch) = boochar[TRUE]);
    colcalc := (gchar(s, 2, ch) = boochar[TRUE]);
    i := 3;
    stofmt(s, i, sheetfmt)
  end;
  readsp := status
end;
```

把行的格式自檔案讀進，寫出也是簡單的，要存行的格式，只要使用人使用格式命令即可，同時，我們尚需儲存資料，以便告訴我們，那些行有特別的格式。

完成這個部份的方法有很多，我們的方法是，先寫一個63——字元的字串到檔案，告訴它那一行有特別的格式（每一表報行一個字元）。如第 *i* 表報行有特別格式，字串的第 *i* 個字元就是 **<N>**；否則就是 **<Y>**。於是乎，若 1 , 2 , 5 , 6 , 和 10 都有特別格式，字串就成

NNYYNNYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY

在這字串寫到輸出檔案，就將字串代表每一特別的格式，一行

• 8 電子工作表格 •

代表一個，上例中要寫出 5 行，每一個在 63 一字元字串中對應一個〈N〉。

writocols 函數如下：

```
function writocols(var f: tfile; var tf: tfrec; var status: iostatus): iostatus;
{ Write column data to file; assumes MAXCOLS < MAXSTR }

var
  i: integer;
  s: string;
begin { writocols }
  s := '';
  for i := 1 to MAXCOLS do
    addchar(s, boochar[usesheetfmt[i]], MAXSTR);
  addchar(s, chr(13), MAXSTR);
  if tfwrite(f, tf, s, status) = GOODIO then begin
    i := 1;
    while (i <= MAXCOLS) and (status = GOODIO) do begin
      if not usesheetfmt[i] then begin
        fttos(colfmt[i], s);
        addchar(s, chr(13), MAXSTR);
        status := tfwrite(f, tf, s, status)
      end;
      i := i + 1
    end
  end;
  writocols := status
end;
```

readcol 常式首先讀進 63 一字串，以顯示那一行有特的格式。此後，對於每一個特別的，被格式化的行（在字串中的每一〈N〉字元），**roadcols** 皆自包含行的格式字串檔案讀進一行。

readcols 函數如下：

```
function readcols(var f: tfile; var tf: tfrec; var status: iostatus): iostatus;
{ Read column data }

var
  s: string;
  i, j: integer;
  ch: char;
begin { readcols }
  if tfread(f, tf, s, MAXSTR, status) = GOODIO then begin
    for i := 1 to MAXCOLS do
      usesheetfmt[i] := (gchar(s, i, ch) = boochar[TRUE]);
    i := 1;
    while (i <= MAXCOLS) and (status = GOODIO) do begin
      if not usesheetfmt[i] then
        if tfread(f, tf, s, MAXSTR, status) = GOODIO then begin
```

```

        j := 1;
        stofmt(s, j, colfmt[i])
      end;
      i := i + 1
    end;
  end;
  readcols := status
end;

```

現在考慮每一非——空格資料的讀進或寫到磁碟。對於每一格的座標都要儲存，它顯示欄位，有一些指示一格是一公式，或一標記。如果格包括一個公式，我們也同樣要存 **forminfo** 資料：公式、格的值，計算的錯誤狀態，和格式資訊。

以上有很多不同的方式可以解決，我們選擇下個方法：每一非——空格就由本文檔的二或三行所代表，只有在格包含公式時，需要 3 行；第 3 行就是字串本身。

第 2 行印格的顯示行。

第 1 行包含存格資訊，而本行的第一項目是格的位置，形式是【<column> , <row>】，這個座標後立即跟著一個字元，它指示格是否包括一個標記或一公式：<L> 表標記，<F> 表公式。

如果格包含標記外，就沒有任何其他資訊。如格包含一公式，在<F>後的字元是 **cellstat** 欄，如 **cellstat** 為 **OK**，表示是一空白字元；否則就是一個指示錯誤的字元：<O> 表溢位，<u> 表超下限，而<Z> 表除以零。

接下來出現格——格式資訊。如果，**usecolfmt** 欄是 **TRUE**（格利用缺段格式值），<Y> 字元就放在下一個位置。若 **usecolfmt** 是 **FALSE**（使用人賦予該格一特別的格式，下一個位置，就放置一個<N> 字元。再接著是字串，代表格式參數和一逗號，來區分格式字串與後續資料。最後，格的值被附加到字串，第一個值是 **cellval.expo**，再來是逗號，其次是 **cellval.frac**。

爲了解其工作，請看下例。

最簡單的案例是一格包括一標記，若格在【10,5】包括標記<Yearly Sales>，存在檔案的應該是
(10,5) L

rearly sales

一格包括一公式就比較複雜了。假設格在【3,23】包括公式SUM(【+0,-10】:【+0,-2】)，其值爲1223.45，同時假設格使用缺設行格式F10.2，則下列三行要被寫到本文檔案中：

```
[3,23]F Y68,122345000000  
1223.45  
SUM([+0,-10]:[+0,-2])
```

本文第一行包括座標【3,23】，<F>表示格包括一公式，一空白指示 calstatus 欄是OK，<Y>表示格使用缺設的行格式，格的數值由 xreal 值的 frac 和 expo 欄指示之（ F I X S I Z E 此處假定爲 12 ）。

如果不使用缺設格式，格有它自己的格式參數，這參數被插進本文檔的第一行。假設，格的格式被註明爲F8.2，則本文的第一行是

```
[3,23]F NF8.2,68,122345000000
```

本文的第二行的顯示欄也改變，以反映新的格式。

翻譯和寫一個格的資料，到輸出檔，由 writecell 函數處理：

• 高等 Pascal 程式設計技巧 •

```

function writecell(var f: tfile; var tf: tfile; cp: cellptr;
                  var status: iostatus): iostatus;
{ Write cell data to file }

var
  s1, s2, s3: string[81];      { MAXSTR + 1 -- allow room for CR at end }
  s: string;

begin { writecell }
  with cp^ do begin
    s2 := '';
    s3 := '';
    ctos(cellcol, cellrow, FALSE, FALSE, s1);
    if display <> NIL then
      s2 := display^;
    if fp = NIL then { label }
      addchar(s1, 'L', MAXSTR)
    else { formula }
      with fp^ do begin
        addchar(s1, 'F', MAXSTR);
        addchar(s1, statchar[cellstat], MAXSTR);
        addchar(s1, boochar[usecolfmt], MAXSTR);
        if not usecolfmt then begin
          fttos(cellfat, s);
          s1 := concat(s1, s, ',');
        end;
        itos(cellval.expo, 0, s);
        s1 := concat(s1, s, ',');
        ftostr(cellval.frac, 0, 0, s);
        s1 := concat(s1, s);
        if formula <> NIL then
          s3 := formula^
        end;
        addchar(s1, chr(13), MAXSTR);
        addchar(s2, chr(13), MAXSTR);
        addchar(s3, chr(13), MAXSTR);
        if tfwrite(f, tf, s1, status) = GOODIO then
          if tfwrite(f, tf, s2, status) = GOODIO then
            if fp <> NIL then
              status := tfwrite(f, tf, s3, status)
            end;
        writecell := status
      end;
end;

```

readcell 保留該過程，自本文檔讀資料，並在特定位置上，建立一個新格：

```

function readcell(var f: tfile; var tf: tfile; var status: iostatus): iostatus;
{ Read cell data from file }

var
  s1, s2, s3: string[81];      { MAXSTR + 1 -- allow room for CR at end }
  i, col, row: integer;
  success, junk: boolean;
  c: char;
  cp: cellptr;

begin { readcell }
  if tftread(f, tf, s1, MAXSTR + 1, status) = GOODIO then begin
    i := 1;
    stoc(s1, i, col, row, junk, junk);

```



```

if (col>=1) and (col<=MAXCOLS) and (row>=1) and (row<=MAXROWS) then begin
  cp := findcell(col, row);
  if cp <> NIL then
    erasecell(cp);
  success := storecell(col, row, cp);
  if success then
    with cp^ do
      if tftread(f, tf, s2, MAXSTR + 1, status) = GOODIO then begin
        chopchar(s2);
        if s2 <> '' then
          success := storestr(s2, display);
        if success and (gchar(s1, i, c) = 'F') then begin { formula }
          success := storeform(fp);
          if success then
            with fp^ do begin
              if gchar(s1, i + 1, c) = statchar[OK] then
                cellstat := OK
              else if c = statchar[OVERFLOW] then
                cellstat := OVERFLOW
              else if c = statchar[UNDERFLOW] then
                cellstat := UNDERFLOW
              else
                cellstat := ZERODIVIDE;
              usecolfmt := (gchar(s1, i + 2, c) = boochar[TRUE]);
              i := i + 3;
              if not usecolfmt then begin
                stofmt(s1, i, cellfmt);
                if gnbchar(s1, i, c) = ',' then
                  i := i + 1;
              end;
              cellval.expo := setparam(s1, i, MINEXP, MAXEXP, MINEXP);
              if gnbchar(s1, i, c) = ',' then
                i := i + 1;
              junk := stof(s1, i, 0, cellval.frac);
              if tftread(f, tf, s3, MAXSTR+1, status) = GOODIO then begin
                chopchar(s3);
                if s3 <> '' then
                  success := storestr(s3, formula)
              end
            end
          end
        end
      end
    end
  if not success then begin
    memout;
    status := ENDFILE
  end
end
end;
readcell := status
end;

```

你可能希望用不同的方法，來儲存和接達表報資料，在讀／寫方面的時間方面改進；在磁碟上儲存表報資料檔使用的空間減少。

排印表報 (Print the sheet)

最後一個命令是排印命令（**</P>**），使用者鍵入這個命令，就可以把表報送到列表機排印。在鍵入**</P>**命令後，**Pascalc** 即詢問：

Print sheet. Are you sure?(Y/N):

如果使用者回答**<Y>**，本程式立刻給一個「排印參數」，它與第五章的 **print** 程式很像，使用者可設定各排印頁次的行數，和設定初值，和終止送往列表機，定左邊和右邊的邊緣。此外，尚可根據使用者鍵入表報的部份（左上角和右下角座標）來排印特定部份表報。

由於使用者的意願，而改變參數，表報的部份被排印，工作表報重新被顯示。

本程式的部份，重用很多第五章的 **print** 程式之代碼，茲列出 **doprint** 的模組——巢式引數如下：

```
doprint
  changepps
  printsheet
    decode *
      gesc *
        htoc*
    initprinter *
    termprinter *
    lskip
    printrow
      lprint *
```

要重新寫的常式只有：**doprint**，**changepps**，**printsheet**，**lskip**，和**printrow**；其餘的常式，都在第五章設計好了。**doprint** 如下：

```
{Sv-}
procedure doprint;
( Print sheet )

const
  LSTRSIZE = 255;
```

• 8 電子工作表格 •

```

type
  longstring = string[255];

{-----}
{ Modules to be inserted here: }
{      changepps      }
{      printsheet     }
{-----}

begin ( doprint )
  if ask('Print sheet. Are you sure?(Y/N):', inline, FALSE, TRUE) then begin
    changepps;
    printsheet;
    drawsheet;
    labelcols(firstcol);
    labelrows(firstrow);
    disp sheet(firstcol, firstrow)
  end
end;
{$v+}

```

changepps 如下：

```

procedure changepps;
( Change print parameters )

var
  margin: integer;
  s1, s2: string;

begin ( changepps )
  crt(CLEAR);
  margin := (MAXCRTCOL - 39) div 2;
  center('Print Parameters', 2);
  itos(pagelen, 0, s1);
  posstr(concat('Page length (lines):', s1), margin, 4);
  itos(lmarg, 0, s1);
  posstr(concat('Left margin:', s1), margin, 6);
  itos(rmarg, 0, s1);
  posstr(concat('Right margin:', s1), margin + 17, 6);
  ctostr(printx1, printy1, FALSE, FALSE, s1);
  posstr(concat('Upper left corner:', s1), margin, 8);
  ctostr(printx2, printy2, FALSE, FALSE, s1);
  posstr(concat('Lower right corner:', s1), margin, 10);
  posstr(concat('Printer init. string:', pinit), margin, 12);
  posstr(concat('Printer term. string:', pterm), margin, 14);
  while ask('Any changes?(Y/N):', MAXCRTROW - 1, FALSE, TRUE) do begin
    pagelen := getint(margin + 20, 4, MTOP+MBOT+1, MAXPLEN, pagelen, TRUE);
    lmarg := getint(margin + 12, 6, 0, LSTRSIZE - 1, lmarg, TRUE);
    rmarg := getint(margin + 30, 6, lmarg + 1, LSTRSIZE, rmarg, TRUE);
    getacc(margin + 18, 8, 1, MAXCOLS, printx1, 1, MAXROWS, printy1, printx1,
      printy1);
    getacc(margin + 19, 10, printx1, MAXCOLS, printx2, printy1, MAXROWS,
      printy2, printx2, printy2);
    getstring(pinit, PCOMSIZE, margin + 21, 12, pinit, [' '..'^'], FALSE);
    getstring(ptermin, PCOMSIZE, margin + 21, 14, pterm, [' '..'^'], FALSE)
  end
end;

```

在設定參數之後，doprint 呼叫 printsheet：

• 高等 Pascal 程式設計技巧 •

```

procedure printsheet;
{ Print specified region of sheet }

const
    PRINTER = 6;          { Unit number of printer }

var
    status: iostatus;
    r, lineno: integer;

{-----}
{ Modules to be inserted here: }
{      decode           }
{      initprinter      }
{      termprinter      }
{      lskip            }
{      printrow         }
{-----}

begin { printsheet }
    if initprinter(pinit, status) = G000I0 then begin
        lineno := 1;
        r := printy1;
        while (status = G000I0) and (r <= printy2) do begin
            if lineno <= 1 then begin
                status := lskip(MTOP, status);
                lineno := lineno + MTOP;
            end;
            if status = G000I0 then begin
                status := printrow(r, status);
                lineno := lineno + 1;
                r := r + 1;
            end;
            if (status = G000I0) and (lineno >= pagelen - MBOT) then begin
                status := lskip(pagelen - lineno + 1, status);
                lineno := 1;
            end;
        end;
        if (lineno > 1) and (status = G000I0) then
            status := lskip(pagelen - lineno + 1, status);
        status := termprinter(pterm, status);
    end;
end;

```

printsheet 是最複雜不過了，它定好列表機的初值，再呼叫 **printrow** 來排印特定範圍的每一列，其他的常式，僅在右方加上頁次斷點。表報排印後，**printsheet** 呼叫 **lskip**，促使列表機調整報表紙到下頁的頂點，如需繼續排印任何字串，**termprinter** 就把終止字串送到列表機去。

lskip 常式也簡單：

• 8 電子工作表格 •

```
function lskip(n: integer; var status: iostatus): iostatus;
( Send n carriage returns to printer )

var
  ch: char;

begin { lskip }
  status := GOODIO;
  ch := chr(13);
  ($I-)
  while (n > 0) and (status = GOODIO) do begin
    unitwrite(PRINTER, ch, 1);
    status := ioresult;
    n := n - 1
  end;
  ($I+)
  lskip := status
end;
```

真正的工作在 **printrow** 完成，它把 CRT 的動作，一列一列的排印，它的邏輯與 **dispro** 類似：對列的每格都計算，看看是否在特定的左邊和右邊之間，或是被切割到其中的一端。

```
function printrow(r: integer; var status: iostatus): iostatus;
( Print one row of sheet )

var
  s: longstring;
  cp: cellptr;
  done: boolean;
  c1, c2, ptrcol, ls: integer;

{-----}
{ Modules to be inserted here: }
{      lprint      }
{-----}

begin { printrow }
  ls := 0;
  ($r-)
  fillchar(s[1], rmarg, ' ');
  cp := rowptr[r];
  done := FALSE;
  while (cp <> NIL) and not done do begin
    with cp^ do begin
      ptrcol := colpos[cellcol] - colpos[firstcol] + lmarg + 1;
      if (cellcol > printx2) or (ptrcol > rmarg) then
        done := TRUE
      else if display <> NIL then begin
        c1 := max(1, lmarg - ptrcol + 2);
        c2 := min(length(display^), rmarg - ptrcol + 1);
        if c2 > c1 then begin
          moveleft(display^[c1], s[ptrcol], c2 - c1 + 1);
          ls := max(ls, ptrcol + c2 - c1)
        end
      end
    end;
    cp := cp^.rightptr
  end;
end;
```

```
if ls > 0 then begin
  s[0] := chr(ls);
  status := lprint(s, status)
end;
{$r+}
if status = GOODIO then
  status := lskip(1, status);
printrow := status
end;
```

字串 S 將送到列表機的行數累積起來，而字串流程長度保存在變數 ls 中，在列的每一格檢查完畢（或全部到右邊），lprint 即排印 s。

建議 (Suggestions)

pascalc 是一個大程式，程式愈大占用空間愈大，有改進的必要，尤其近乎是交錯程式 (interactive program)，運轉的愈快愈好。如果能改進 **calc**，則對 **pascalc** 甚有助益，在 CRT 上已提出了高等功能，諸如插入和刪除，這些都可以在速度上改進。

更有效的使用記憶體，就可允許 **pascalc** 像以前一樣處理大量的表報。

有一特殊建議：寫一版 **pascalc**，當使用人第一次進入時，即「預先解譯」。pascalc 花費太多的時間去重新計算鍵入的公式，包括在 **stox** 和 **stoc** 常式內轉換。預先解譯公式，也要做轉換的工作，不過它儲存公式的真正的數字，而非存串來代表公式。

將公式重新安排成後置記法 (postfix notation) 形式，就很容易評估一個非——遞迴常式。（如你不熟悉這部份請參閱下節的建議）

可以在 **pascalc** 內增加一些特性（平方根函數，三角函數，對數，指數等），使其更具彈性，日曆的算術也可加上去。

你也可依不同的應用設計不同版的 **Pascal**，在用於統計工作，計算用的內建函數，標準離差，**chi-square** 等，一個工程師可能希望一個內建矩陣，反矩陣函數。經理則希望評估時間序列，諸如搬移的平均數、線性迴歸 (**liner regression**) 等。

彈性格式可能有用：在一行內可以加逗號，以「浮點」錢號，百分號等，此外尚可增加分類命令，自動的按照字母或數字次序排列。

其他要增加的特性是增加 **pascal** 的簡易性。例如，改變這個版本的公式或標記的唯一方法，是重打整個公式或標記，增加 **getstring** 或在 **pascal** 內當作命令來用，就比較好。更允許用助憶 (**mnemonic**) 符號表示格，例如，在 [5 , 1] 之格包含一個利率，使用人即可將格命名為 **INTRATE**，在公式內使用這個識別字 (**identifier**) 而不用格的座標。

每個挑戰性的方案，皆可用一個 “undo” 命令實行之，它會保留使用人的最後一個命令。

這些建議都是可能改進的表面的草稿。

推薦閱讀 (Recommended Reading)

變動記憶管理的不同方法，見 D.Knuth 1973 著 **Fundamental Algorithm**。本書 **alloc** 和 **free** 用到的粒子演算，參考 B.Kernighan 和 D.Ritchie (**prentice-Hall**, 1978) 著的 **C 程式語言**。

稀疏矩陣，先進先出表列，環式緩衝器，和後置記法，都在資料結構的書中討論，諸如 Knuth 著書和 A.Tenenbaum 和 M.Augenstein (**Printice-Hall**, 1981) 著 **Data structures Using Dascal**。和 R.Baron (**Van Nostrand Reinhold**,

• 高等Pascal 程式設計技巧 •

1980) 合著的Data structure and ther Implementation。

APPLE PASCAL的型態——前置資訊，取自Attach
—BIOS (B.Haynes 著)。這文件包含APPLE PASCAL
輸出／入系統。

程式設計的哲學和方法，包括測試，見E.yoardon 1975
年著「程式結構和設計的技術」。(Techniques of progr
am structure and Design)。

附錄 A

易傳性問題的探討與對策

標準 PASCAL 語言用來撰寫易傳性程式 (Portable Programs) 比較困難。所謂易傳性程式乃是指該程式幾乎不必經過修改，即可在各種電子計算上運轉。而本書設計程式的時候，爲了提高程式的簡潔性、有效性、和實用性，所以對於程式的易傳性，做了某些程度的犧牲。是以這些程式並非全然易傳性的，他們運用了許多你可能無法使用於 APPLE PASCAL 1.1 版本的非標準特性。

本附錄特別針對所用過的一些非標準化的 APPLE PASCAL 特性，來說明如何用其他種類的 PASCAL 撰寫，我們也將盡可能的對各種 PASCAL 語言，提供有關易傳性問題的解決方案。至於某些無法做到易傳性的部份，我們也至少提供一些助益。

字串 (STRINGS)

我們所有的程式，都使用 APPLE 或 UCSD PASCAL 的內建字串 (built-in string)。不幸的是標準 PASCAL 並未提供通用的字串處理常規。

易傳性的表示法 (Portable Representation)

定義一個易傳性的字串資料型態的方法有很多，我們不只需要表示出字串的字元本身，既然我們希望字串能擁有各種不同的字元，也得同時表示出在這個字串裡，每個字串現有多少字元。

我們的解決方法是：定義這個字串資料，一方面是一筆內含字串長度的記錄；另一方面是一個字串內含有字元的陣列。

```
<type>
  string = packed record
    length: 0..MAXSTR;
    c: packed array [1..MAXSTR] of char
  end;
```

在這種表示方法之中，一個字串變數「S」的長度是 `s.length`。（因此，我們可用 `s.length` 表示，以取代APPLE/UCSD PASCAL 語言中的呼叫功能 `length(s)`）。本字串中的第 `i` 個字元是 `s.c[i]`。本字串最多有 `MAXSTR` 個字元。（`MAXSTR` 為預先定義好的整常數。）

輸出 (Output)

我們不在使用內建常規 `write`，來顯示一個字串。標準 PASCAL 語言的 `write` 和 `writeln` 只能處理實數、整數、字元、布耳數、和定長的縮緊式陣列（`fixed-length packed arrays of characters`）。因此，我們需要一個特殊的常規，一次一個字元的輸出一個字串。

```
procedure writestr(var s: string);
{ Write string }

var
  i: integer;

begin { writestr }
  with s do
    for i := 1 to length do
      write(c[i])
  end;
```

指派 (Assignment)

當我們定義字串型態為記錄 (record) 時，在程式裡並沒有易傳性的方法能來說明一個特定字串常數。我們無法直接指派 (assignment) 一個字串常數給字串變數，這是與 AP-
PLE 及 UCSD PASCAL 語言的不同之處。例如，如果 S 是一個事先定義過的字串變數，不能只寫：

```
s := 'Colette';
```

(在標準 PASCAL 中，這種指派只有在 S 如下列方式預先定義才能有效：

```
<var>  
s: packed array [1..7] of char; { 7 = # of characters  
                                in 'Colette' }
```

唯此時 S 恰好七個字元，其長度不能有多於七、或少於七的情形。)

在無法直接使用指派情形之下，我們就只能用下述方法來指派字串：

```
...  
with s do begin  
  length := 7;  
  c[1] := 'C';  
  c[2] := 'o';  
  c[3] := 'l';  
  c[4] := 'e';  
  c[5] := 't';  
  c[6] := 't';  
  c[7] := 'e';  
end;  
...
```

然而，這個方法過於龐大而且笨拙，字串愈長其情況愈糟。

比較 (Comparisons)

直接指派與標準關係運算元 (standard relational operations (< , > , = , 等等) 相比較 , 同樣無法適用易傳性的方法。在比較兩個字串時 , 必須另寫一常規 `strcmp` 來接受兩個字串為引數 (arguments) , 當兩字串相等時為 0 , 當前者小於後者時為 - 1 , 前者大於後者時為 + 1 。

```
function strcmp(var s1, s2: string): integer;
{ Compare strings; return -1,0,+1 if s1 < , = , > s2 }

var
  i: integer;
  c1, c2: char;

begin { strcmp }
  i := 0;
  repeat
    i := i + 1;
    c1 := gchar(s1, i, c1);
    c2 := gchar(s2, i, c2)
  until (c1 <> c2) or (c1 = chr(0));
  if c1 = c2 then { Strings are equal }
    strcmp := 0
  else if c1 < c2 then
    strcmp := -1
  else
    strcmp := 1
end;
```

為進一步瞭解其使用狀況 , 可參考下面的一個以 APPLE PASCAL 語言來直接比較字串的程式片斷。(摘自本書第四章的 `insertnode`)。

```
...
if id < p^.name then
  insertnode(p^.left, id, x)
else if id > p^.name then
  insertnode(p^.right, id, x)
else
  p^.value := x
...
```

• 附錄 A 易傳性問題的探討與對策 •

如果使用 **strcmp** 來取代非易傳性的直接比較方式，可將代碼改為：

```
...
case strcmp(id, p^.name) of
  -1:
    insertnode(p^.left, id, x);
  0:
    p^.value := x;
  1:
    insertnode(p^.right, id, x)
end
...
```

字串處理常規 (String-Manipulation Routines)

我們必須改寫任何非易傳性的字串處理常規，來取代易傳性的字串處理表示法。現在舉一個 **gchar** 新版本的實例如下：

```
function gchar(var s: string; i: integer; var ch: char): char;
{ Extract character from string }

begin { gchar }
  with s do
    if (i < 1) or (i > length) then
      ch := chr(0)
    else
      ch := c[i];
    gchar := ch
  end;
end;
```

我們的 **addchr** 常規，在字串的後面附加一字元的方式，也得修改如下：

```
procedure addchar(var s: string; ch: char; maxlen: integer);
{ Add character to end of string }

begin { addchar }
  with s do
    if length < maxlen then begin
      length := length + 1;
      c[length] := ch
    end
  end;
end;
```

一個新版的 **chopchar**，它刪除了字串的最後一個字元，也容易寫成：

```
procedure chopchar(var s: string);
{ Delete character from end of string }
begin { chopchar }
  with s do
    if length > 0 then
      length := length - 1
end;
```

除了重寫字串的處理常規之外，還必須設計內建式的APPLE/UCSD PASCAL 語言的字串處理常規。例如，序連（concatenate）兩個字串的易傳性 **concat**，其撰寫方式如下：

```
procedure concat(var s1, s2, s3: string);
{ Concatenate s1 and s2 into s3 }

var
  i: integer;

begin
  s3 := s1;
  with s2 do
    for i := 1 to length do
      addchar(s3, c[i], MAXSTR)
end;
```

本例，將字串 **S1** 與 **S2** 結合成字串 **S3**，並將 **S3** 字串以 **Var** 變數型態傳回呼叫常規。APPLE/UCSD PASCAL 的 **concat** 較易傳型的略為簡單些。它可接受任何的字串引數（string arguments），並把字串序連的函數結果（function result）傳回來。遺憾的是，這些特性都無法應用於易傳的 PASCAL 常規。

我們也使用 APPLE/UCSD 內建的 **copy** 常規，自字串中擷取子字串（substring）。這裡有一個易傳型的實例如下：

```
procedure copy(var s: string; i, n: integer; var substr: string);
{ Copy n characters starting at ith character of s into substr }

var
    last: integer;

begin { copy }
    substr.length := 0;
    i := max(1, i);
    last := min(i + n - 1, s.length);
    while i <= last do begin
        addchar(substr, s.c[i], MAXSTR);
        i := i + 1
    end
end;
```

此種 **copy** 由字串 **s** 的第 **i** 個字元開始，移動 **n** 個字元到 **substr** 字串。同樣的，APPLE/UCSD PASCAL 的 **copy** 亦較為簡易。它只需將其函數結果（function result），亦即子字串傳回即可。

一個易傳性的 **pos** 並不難寫，它不過是自一子字串中，搜尋一子字串的常規。

```
function pos(var pat, s: string; i: integer): integer;
{ Search for occurrence of pat in s starting at ith character of s }

var
    j, k: integer;
    found: boolean;
    c1, c2: char;

begin { pos }
    found := FALSE;
    while (gchar(s, i, c1) <> chr(0)) and not found do begin
        j := i;
        k := 1;
        c2 := gchar(pat, k, c2);
        while (c2 <> chr(0)) and (c1 = c2) do begin
            j := j + 1;
            k := k + 1;
            c1 := gchar(s, j, c1);
            c2 := gchar(pat, k, c2)
        end;
        if c2 = chr(0) then
            found := TRUE
        else
            i := i + 1
    end;
    if found then
        pos := i
    else
        pos := 0
end;
```

本常規由字串 **s** 中的第 **i** 個字元開始，搜尋 **pat** 字串。（這種呼叫順序的用處，比 APPLE/UCSD PASCAL 中的 **pos** 大，因為後者只能由 **s** 字串的第一個字元開始搜尋。）如果，在 **s** 字串中找到 **pat** 字串，**pos** 將指標傳回（即 **pat** 字串在 **s** 字串的開始的位置。）否則，**pos** 將傳回 0。

總結 (Summary)

顯而易見的，我們已盡全力去發展易傳性的字串常規，就如同本書所採用的內建式 APPLE/UCSD PASCAL 的字串型態一般。然而，這種易傳性字串表示法，仍然有許多弊端。除了無法明確地給予一字串定初值之外，尚且不能把字串常數「易傳地」傳送給程序（procedure）和函數（function）。舉例而言，在 APPLE/UCSD PASCAL 語言中，使用下法是合理的：

```
...
center('Press <RETURN> to continue.', MAXCRTROW)
...
```

若使用易傳性的字串表示法，必須寫成：

```
...
with tempstr do begin
  length := 27;
  c[1] := 'P';
  c[2] := 'r';
  c[3] := 'e';
  ...
  c[25] := 'u';
  c[26] := 'e';
  c[27] := '.';
end;
center(tempstr, MAXCRTROW)
...
```


當然，tempstr 必須被宣告成一個字串變數才行。

易傳性的另一個缺點；就是儲存的空間的需求上缺乏彈性（inflexibility）。MAXSTR 常數必須依據字串最大可能的長度來定義。如果某一程式擁有各種不同的長度的字串，在儲存較短的字串上，將會導致許多記憶空間不必要的浪費。反之，APPLE/UCSD PASCAL 語言，允許程式設計師可以根據不同的最大的可能長度，來定義字串型態不同的特性，或多或少減輕了這個問題的嚴重性，茲說明如下：

```
<type>
  identifier = string[IDSIZE];
  word = string[MAXWORD];
  longstring = string[LSTRSIZE];
```

在APPLE和UCSD PASCAL 語言之中，這種不同型態的字串，可以透過字串處理常規來工作。然而在標準的PASCAL 語言裡，絕對不允許字串型態有任何混亂。

此外，尚有一種解決問題的辦法，那就是ISO的標準PASCAL調合陣列（conformat array）。這是wirth最原始的PASCAL語言所附加的一種非必須的特性。（所謂「非必須」，係指在PASCAL中可以省略掉，却仍然符合ISO標準的。），這種特性，提供了我們撰寫通用常規，來處理不同大小的陣列，甚至字元陣列的能力，然而有許多PASCAL語言，並未具備這種特性，但如果你的PASCAL語言有，你將如何充分利用？

總而言之，大多數的PASCAL語言，都提供了一些非易傳性，類似字串的資料型態，作為標準PASCAL語言的延伸（extension）。作者以為與其忍受易傳性程式的缺點，遠不如修改本書的程式，使其適合你所用的PASCAL語言的字串型態來得有效。

長整數 (LONG INTEGERS)

APPLE和UCSD PASCAL所提供的另一種有效，但不是標準的資料型態、定義如下：

```
<type>
  fixed = integer[FSIZE];
```

此處，FSIZE係指一個依據數位最大長度，來設定的常數。我們這裡說明的主要原因是告訴大家，如何在兼顧易傳性的原則下，得到較高精確度資料型態的好處。

* 易傳性表示法 (Portable Representation)

我們可以有各種不同的方法，來表示一個長整數 (long integer)。這裡所介紹的是一種儲存一陣列中各個十進位數的方法，茲說明如下：

```
<type>
  fixed = record
    sign: -1..1;
    digit: packed array [1..FSIZE] of 0..9
  end;
```

在這種表示法中，固定變數 (fixed variable) *f*，如為負數，其正負號欄 (sign field) 為 -1；如果是0、或是正數，其正負號欄值為 +1。*f.digit[1]* 是一數，代表單位數位，*f.digit[2]* 是十數位，餘類推……。一個聰明的編譯器 (compiler)，會將這些數位，以二進碼十進數 (BCD——binary coded decimal) 表示法聚集起來，每一個數元組 (byte) 中，儲存二個數位欄 (digit field)。

指派表示法 (Assignment Representation)

就和易傳性字串一樣，對於固定資料型態的變數作直接指

派，也是不可能的，在APPLE/UCSD PASCAL 語言中，我們可以寫成：

```
f := 314159;
```

在易傳性表示法中，對數位必須逐次指派：

```
with f do begin
  sign := 1;
  digit[1] := 9;
  digit[2] := 5;
  digit[3] := 1;
  digit[4] := 4;
  digit[5] := 1;
  digit[6] := 3;
  for i := 7 to FIXSIZE do
    digit[i] := 0
end;
```

(此處假設FIXSIZE至少為6)

轉換性 (Conversions)

這種處理需要新版的ftos (把一個固定數轉換為一字串) 和新版的ftos (將一字串轉換為一固定數)。首先，先介紹新版的ftos，必須注意的是我們仍保留了可在一數目中的某一特定位置，加入十進位小數點的選擇權：

```
procedure ftos(var f: fixed; width, ndigs: integer; var s: string);
  (Convert fixed to string)

var
  nc, i, j: integer;
  ch: char;

begin (ftos)
  with s, f do begin
    length := 0;
    i := FIXSIZE;
    while (i > 1) and (digit[i] ≠ 0) do
      i := i - 1;
    nc := 0;
    repeat
      nc := nc + 1;
      addchar(s, chr(digit[nc] + 48), MAXSTR);
      if nc = ndigs then
        addchar(s, '.', MAXSTR)
    until (nc ≥ i) and (nc > ndigs);
```

```
if sign < 0 then
  addchar(s, '-', MAXSTR);
while length < width do
  addchar(s, ' ', MAXSTR);
i := 1;
  j := length;
  while i < j do begin
    ch := c[i];
    c[i] := c[j];
    c[j] := ch;
    i := i + 1;
    j := j - 1;
  end
end
end;
```

這種 **ftos** 的版本假設字串仍是易傳性的；是以改編 **ftos** 爲它種的字串表示法，應是相當簡單的。

下面的 **stof** 函數，也只需對舊版的略加修改即可：

```
function stof(var s: string; var i: integer; ndigs: integer; var f: fixed):
boolean;
{ Convert string to fixed }

var
  j, nsig, nafter: integer;
  c: char;

begin { stof }
  with f do begin
    for j := 1 to FIXSIZE do
      digit[j] := 0;
    sign := 1;
    nsig := 0;
    nafter := 0;
    j := 1;
    if gnbchar(s, i, c) in ['+', '-'] then begin
      if c = '-' then
        sign := -1;
      i := i + 1;
    end;
    while gnbchar(s, i, c) = '0' do
      i := i + 1;
    while gnbchar(s, i, c) in ['0'..'9'] do begin
      if nsig < FIXSIZE then
        digit[FIXSIZE - nsig] := ord(c) - 48;
      nsig := nsig + 1;
      i := i + 1;
    end;
    if c = '.' then begin
      i := i + 1;
      if nsig = 0 then
        while gnbchar(s, i, c) = '0' do begin
          nafter := nafter + 1;
          i := i + 1;
        end;
      while gnbchar(s, i, c) in ['0'..'9'] do begin
```

```
        if (nsig < FIXSIZE) and (nafter < ndigs) then
            digit[FIXSIZE - nsig] := ord(c) - 48;
            nsig := nsig + 1;
            nafter := nafter + 1;
            i := i + 1;
        end
    end;
    if (nafter < ndigs) and (nsig > 0) then
        while (nsig < FIXSIZE) and (nafter < ndigs) do begin
            nsig := nsig + 1;
            nafter := nafter + 1;
        end;
    if (nsig < FIXSIZE + nafter - ndigs) and (nsig > 0) then begin
        for j := 1 to nsig - nafter + ndigs do
            digit[j] := digit[j] + FIXSIZE - nsig + nafter - ndigs;
        for j := nsig - nafter + ndigs + 1 to FIXSIZE do
            digit[j] := 0;
        end
    end;
    stof := (nsig - nafter) <= (FIXSIZE - ndigs)
end;
```

此外，大家可以自己練習寫其他的轉換常規：諸如固定數轉換成整數，整數轉換為固定數，固定數轉換成實數……等等；其中沒有一個是困難的。

比較表示法 (Comparisons)

APPLE 和 UCSD PASCAL 語言中，運用關係運算子，可直接比較長整數；和字串一樣的是，我們仍然必須撰寫一個獨立的常規，來達到易傳的目的：

```
function fixcomp(var f1, f2: fixed): integer;
( Compare fixed numbers, return -1,0,-1 if f1 <,> f2 )

var
    i: integer;
    done: boolean;

begin ( fixcomp )
    if f1.sign <> f2.sign then
        fixcomp := f1.sign
    else begin
        i := FIXSIZE;
        repeat
            done := (f1.digit[i] <> f2.digit[i]);
            if not done then begin
                i := i - 1;
                done := (i = 0)
            end
        until done;
        if i = 0 then
```

```
        fixcomp := 0
      else if f1.digit[i] > f2.digit[i] then
        fixcomp := f1.sign
      else
        fixcomp := -f1.sign
      end
    end;
  end;
```

加法 (Addition)

APPLE 和 UCSD PASCAL 語言中，利用標準算術算子 (standard arithmetic operations) 可以直接對長整數作的算術運算；這點是十分重要的，因為在易傳性表示法之中，就無法做到。首先，讓我們來看看以下的兩個固定數相加的常規：

```
function fadd(f1, f2: fixed; var f: fixed; var status: xresult): xresult;
( Add two fixed numbers, f := f1 + f2 )

var
  dig, carry, i: integer;
begin ( fadd )
  if f1.sign <> f2.sign then begin
    f2.sign := -f2.sign;
    fadd := fsub(f1, f2, f, status)
  end
  else begin
    f.sign := f1.sign;
    carry := 0;
    for i := 1 to FIXSIZE do begin
      dig := f1.digit[i] + f2.digit[i] + carry;
      f.digit[i] := dig mod 10;
      carry := dig div 10
    end;
    if carry <> 0 then
      status := OVERFLOW
    else
      status := OK;
    fadd := status
  end
end;
```

如果被相加的兩數正負號相反，fadd 將轉換第二位數的正負符號，再呼叫 fsub 來對兩數相減。否則，fadd 將會逐數位的對兩數作相加（就如同你在小學算術中所學到的一樣。）

，並隨時對次數位的進位（**carry**）作記錄。如果**FIXSIZE**的所有數位相加之後，變數**carry**不等於0，那是有溢位（**overflow**）發生。

fadd將任何的錯誤狀況透過**xresult**變數，回轉給呼叫常規；它將會回轉成功（**ok**），或是溢位（**OVERFLOW**）來表示**var**參數**status**和其結果。

減法（**Subtraction**）

fsub常式，比小學算術稍為深奧些，但仍算是簡單的了：

```
function fsub(f1, f2: fixed; var f: fixed; var status: xresult): xresult;
{ Subtract two fixed numbers, f := f1 - f2 }

var
  i, dig, carry: integer;

begin { fsub }
  if f1.sign <> f2.sign then begin
    f2.sign := -f2.sign;
    fsub := fadd(f1, f2, f, status)
  end
  else if f1.sign * fixcomp(f1, f2) < 0 then begin { abs(f1) < abs(f2) }
    fsub := fsub(f2, f1, f, status);
    f.sign := -f.sign
  end
  else begin
    carry := 1;
    f.sign := f1.sign;
    for i := 1 to FIXSIZE do begin
      dig := f1.digit[i] - f2.digit[i] + carry + 9;
      f.digit[i] := dig mod 10;
      carry := dig div 10
    end;
    status := OK;
    fsub := status
  end
end;
```

如果**f1**和**f2**正負相反，**fsub**將轉換**f2**的正負符號，再呼叫**fadd**對**f1**和**f2**加法運算。否則，它將叫**fixcomp**來檢查**f1**和**f2**的絕對值，並比較其大小；如果前者小於後者，**fsub**會再呼叫本身來計算**f2 - f1**，並對其結果**f**的正負號作轉換。

如果上述的兩種情況都未發生，就可運用一般的減法。此

時，將不可能發生溢位（overflow），而且相減的結果之正負號會與 **f1** 和 **f2** 相同。減法也如同加法一樣，由單位數位開始，逐數位地運算。它唯一所有的「非小學」特性，就是變數 **carry**。該變數用於表示一種逆轉借位（inverse borrow），也就是說，如果沒有向前一數位借位的話，**carry** 的值是 1；如果有借位，**carry** 為 0。（如有所懷疑，無妨用手算算看）。

值得注意的是 **fadd** 和 **fsub** 也可以相互呼叫，這表示在一個真正的程式裡，必須宣告成爲一種前向函數（forward function）。

乘法（Multiplication）

當兩個 **n** 數位的數目相乘，結果必定是個 $2n$ 數位或 $(2n - 1)$ 數位的數目。當兩個固定數相乘時（每一個有 **FIXSIZE** 數位），我們可以得到 $2 * \text{FIXSIZE}$ 數位的乘積。如果乘積超過 **FIXSIZE** 數位時，可以有兩種表達方式：一爲回轉固定數的乘積（fixed number），並註明溢位；一爲以一個新的資料型態，也就是一個足以容納所有乘積數位的「較長的」整數來表示。（如此，則溢位情況就不會發生。）

現在以後者爲例來說明。如果你想起了第四章中的 **cale** 程式，我們曾定義一種暫存器（Register）資料型態，來存放計算中間的結果：

```
<type>
  register = integer[REGSIZE];    { REGSIZE = 2 * FIXSIZE + 2 }
```

在易傳性表示法中，作了類似定義如下：

```
<type>
  register = record
    sign: -1..1;
    digit: packed array [1..REGSIZE] of 0..9
  end;
```


這裡，我們將 **REGSIZE** 設定為 $2 * \text{FIXSIZE} + 2$ ，雖然較實際需要的為大，但對由本書第四章的 **xreal** 算術常式，轉換成易傳性的表示法，將頗有助益。

若已給定暫存器型態，來撰寫乘法的常式就相當容易了。

```
function fmul(var f1, f2: fixed; var reg: register; var status: xresult):
                                                    xresult;
{ Multiply two fixed numbers, reg := f1 * f2 }

var
  n1, n2, i, j, carry, dig: integer;
begin { fmul }
  reg.sign := f1.sign * f2.sign;
  for i := 1 to REGSIZE do
    reg.digit[i] := 0;
  n1 := FIXSIZE;
  while (n1 > 1) and (f1.digit[n1] = 0) do
    n1 := n1 - 1;
  n2 := FIXSIZE;
  while (n2 > 1) and (f2.digit[n2] = 0) do
    n2 := n2 - 1;
  for j := 1 to n2 do
    if f2.digit[j] <> 0 then begin
      carry := 0;
      for i := 1 to n1 do begin
        dig := reg.digit[i + j - 1] + f1.digit[i] * f2.digit[j] + carry;
        reg.digit[i + j - 1] := dig mod 10;
        carry := dig div 10;
      end;
      reg.digit[j + n1] := carry;
    end;
  status := OK;
  fmul := status;
end;
```

此處的 **n1** 和 **n2** 分別代表 **f1** 和 **f2** 的有效數位。(我們只需要將二個有效數位相乘即可。)我們以同時運用乘法、加法、在 **reg** 之內直接累積，以取代在小學時所學的那一套，先求部份乘積 (partial products)，再將其相加的方法。

除法 (Division)

與其他的算術常式相比較，建立我們的除法常式 **fdiv** 是比較困難的。然而相同的是，我們仍以小學算術 (此處指的是長除法) 為除法演算的立論基礎。「商」是一次一數位的計算。當算出一「商數位」 (quotient digit) 時，將其與除數相

乘後，來被「被除數」減，減得的結果即為計算下一商數位的「新被除數」。經過一次次的演算，在所有的商數位都已計算完之後，剩餘的被除數就是我們的餘數。

設計除法常式的原則是：以暫存器（register）形態的被除數和固定形態（fixed）的除數為輸入項目；以暫存器（register）的商和固定的（fixed）的餘數為輸出項目。如果有任何錯誤情況發生，也將和以往一樣的，以函數結果和 var 變數的方法，回轉給呼叫程式，但唯一可能的錯誤是除數為 0，因為我們對於輸入和輸出數字大小的選擇，確保了其他異常狀況發生時的適當處理。

在 **fdiv** 的邏輯運算中，我們已將一些運算較單純的「特殊狀況」過濾掉，以強化 **fdiv** 的功能，茲列舉如下：

1. 除數為零時，只需回轉錯誤狀況。
2. 除數數位較被除數為多，則其商為零，而餘數等於被除數。
3. 除數只有一個數位時，只需經過一次除法演算，較正常情況簡易的多。

再以 **fdiv** 的最初的虛擬代碼，來介紹這種結構：

```
begin
  assign signs to quotient and remainder
    (after this we can behave as if dividend and divisor
                                         are positive)
  zero quotient and remainder digits
  count significant digits in dividend (n1) and divisor (n2)
  if divisor is zero
    return ZERODIVIDE error status
  else if n1 < n2 then
    quotient is zero
    remainder := dividend
  else if divisor is single digit (n2 = 1)
    do division directly
  else
    do normal case
end
```

在上述的三種特殊狀況業經適當的處理之假設下，我們可以設定「除數、被除數皆屬二數位以上數目」為正常狀況。除了可預期的簿記之複雜性（確定每一數位都排在適當的位置，正確地處理進位與借位……）外，主要的問題在於如何有故地準確地猜出每一個商數位。（你現在可以試著用手算幾個長除法，就會想起在沒有計算機的日子，猜測過程是如何的費神了。）

我們可以用「以除數的前二數位除目前被除數的前二、三數位」的方式，來對商數做個合理的猜測。（這樣的說法很不清楚，須看 `fdiv` 常式，方知道其精確的做法）。圖 A-1 舉例說明了傳統的長除法如下：

$ \begin{array}{r} 865143 \\ 3142 \overline{) 2718281828} \\ \underline{25136} \\ 20468 \\ \underline{18852} \\ 16161 \\ \underline{15710} \\ 4518 \\ \underline{3142} \\ 13762 \\ \underline{12568} \\ 11948 \\ \underline{9426} \\ 2522 \end{array} $		
	<code>guess := 271 div 31 = 8</code>	
	<code>guess := 204 div 31 = 6</code>	
	<code>guess := 161 div 31 = 5</code>	
	<code>guess := 45 div 31 = 1</code>	
	<code>guess := 137 div 31 = 4</code>	
	<code>guess := 119 div 31 = 3</code>	
	<code>= remainder</code>	

這種猜測的方法下，我們可猜得正確的商數位，或是只比正確的多 1 的商數位。（此項理論的證明，雖不困難，但因不是本書的範疇，故不予介紹。）

藉此猜測法，我們可以撰寫如下的虛擬代碼，以表示除法

常式的「正常部份」。

```
{ normal case }
begin
  calculate number of digits in quotient (nq := n1 - n2 + 1)
    (leading digit of quotient may be zero)
  set v := first two digits of divisor
  for each quotient digit
    set u := (leading two or three digits of dividend)
    calculate guess for quotient digit := u div v
      (if u div v > 9, then guess := 9)
    multiply divisor by guess, subtract from dividend
    if resulting dividend < 0, guess was too big
      guess := guess - 1
      correct dividend by adding back divisor
    set quotient digit to guess
  end-for
  remainder := leftover dividend
end
```

現在，介紹一個完整的 **fdiv** 除法常式：

```
function fdiv(dividend: register; divisor: fixed; var quotient: register;
              var remainder: fixed; var status: xresult): xresult;
{ Divide dividend by divisor, yielding quotient and remainder }

var
  n1, n2, nq, i, j, dig, carry, borrow, guess, u, v: integer;

begin { fdiv }
  quotient.sign := dividend.sign * divisor.sign;
  remainder.sign := dividend.sign;
  for i := 1 to REGSIZE do
    quotient.digit[i] := 0;
  for i := 1 to FIXSIZE do
    remainder.digit[i] := 0;
  n1 := REGSIZE;
  while (dividend.digit[n1] = 0) and (n1 > 1) do
    n1 := n1 - 1;
  n2 := FIXSIZE;
  while (divisor.digit[n2] = 0) and (n2 > 1) do
    n2 := n2 - 1;
  status := OK;
  if (n2 = 1) and (divisor.digit[1] = 0) then { divisor is zero }
    status := ZERODIVIDE
  else if n1 < n2 then { quotient := 0, remainder := dividend }
    for i := 1 to n1 do
      remainder.digit[i] := dividend.digit[i]
  else if n2 = 1 then { Division by single digit }
    while n1 > 0 do begin
      dig := dividend.digit[n1] + 10 * remainder.digit[1];
      quotient.digit[n1] := dig div divisor.digit[1];
      remainder.digit[1] := dig mod divisor.digit[1];
      n1 := n1 - 1
    end
  end
```

•附錄A 易傳性問題的探討與對策•

```
else begin { Normal case }
  nq := n1 - n2 + 1;
  v := 10 * divisor.digit[n2] + divisor.digit[n2 - 1];
  while nq > 0 do begin
    with dividend do
      if n1 < REGSIZE then
        u := 100 * digit[n1 + 1] + 10 * digit[n1] + digit[n1 - 1]
      else
        u := 10 * digit[n1] + digit[n1 - 1];
    guess := min(u div v, 9);
    borrow := 1;
    carry := 0;
    j := nq;
    for i := 1 to n2 do begin
      dig := guess * divisor.digit[i] + carry;
      carry := dig div 10;
      dig := dividend.digit[j] - (dig mod 10) + 9 + borrow;
      dividend.digit[j] := dig mod 10;
      borrow := dig div 10;
      j := j + 1;
    end;
    if u div 100 <= carry - borrow then begin { guess too big }
      guess := guess - 1;
      carry := 0;
      j := nq;
      for i := 1 to n2 do begin
        dig := dividend.digit[j] + divisor.digit[i] + carry;
        dividend.digit[j] := dig mod 10;
        carry := dig div 10;
        j := j + 1;
      end;
    end;
    quotient.digit[nq] := guess;
    nq := nq - 1;
    n1 := n1 - 1;
  end;
  for i := 1 to n1 + 1 do
    remainder.digit[i] := dividend.digit[i];
  status := OK;
end;
fddiv := status;
end;
```

總結 (Summary)

對於字串而言，固定 (fixed) 資料型態的易傳性表示法中最主要的缺點是：無法在表式 (expression) 中使用定常數 (fixed constant)，大量地降低處理速度，以及儲存的缺乏彈性，因而致使寧可使用你自己的 PASCAL 語言所提供的非易傳性、擴增精密度 (extended-precision) 資料型態表示法，不願採行此處所發展的易傳性表示法。

另一方面，這種易傳性表示法可能較你的PASCAL 語言的擴增精密度資料型態，更富彈性。例如，如果你想達到百數位的精密度，只須將**FIXSIZE**設定為100即可。

集合 (Sets)

在本書用到最多的集合，就是字元集合 (sets of characters)。例如，**getkey**使用「合法的」字元集合，也就是說只接受此集合中目前已鍵入的字元，其餘的一律拒絕，我們也可自由運用如下的代碼，來準確地測試一字元是否在某一範圍之內。

```
...
while gnbchar(s, i, ch) in ['0'..'9'] do begin
    ...
    i := i + 1
end;
...

and

...
if ch in ['A'..'F'] then
    ...
```

這些用法均視字元集合型態的合法性而定。但是標準的PASCAL 語言，並不保證字元集合是合法的；只有在字元集合的字元數目不可以大於其最上限，這些數目與實行的方式相關。

如果你的PASCAL 語言不允許使用字元集合，你該怎麼辦呢？在其他情況維持均等之下，轉換為允許使用字元集合的PASCAL 語言，將是你的最佳抉擇。然而，如果你因困於環境轉換環境時，可以採行二種策略。一為避免它們，一為發明

創造它們。

避免使用集合 (Avoiding sets)

在許多情況之下，你可以非常容易地以相當的非集合基礎碼 (non-set-based code) 來取代集合基礎碼 (set-based code)。例如，先前的 **while-loop** 可以改寫如下：

```
...
ch := gnbchar(s, i, ch)
while (ch >= '0') and (ch <= '9') do begin
    ...
    i := i + 1;
    ch := gnbchar(s, i, ch)
end;
...
```

而 **if-test** 可被寫成：

```
...
if (ch >= 'A') and (ch <= 'f') then
    ...
```

然而，這方法並非總是方便、精確的。

創造性使用集合 (Inventing sets)

集合多半用於位元陣列 (bit array)，每一個位元皆用來表示集合內的元素是否存在，我們可以宣告如下：

```
<type>
charset = set of char;
```

而
with

```
<type>
charset = packed array [char] of boolean;
```

許多編譯器，以布耳數緊縮陣列 (packed array of boolean) 為位元陣列，因而促成儲存空間的有效利用。現在，我們寫出如下的代碼：

```
...  
if not(ch in valid) then  
...
```

(假設 `valid` 為 `charset` 型態的變數) 我們可改寫如下：

```
...  
if not valid[ch] then  
...
```

其餘的集合運算，都可以輕易地轉換成這種布耳陣列表示。一如往常，其有下列缺失：無法在原始程式之中表示布耳陣列的常數，致使無法給予一陣列初值。

隨筆 (Miscellany)

在本書的程式中，偶爾使用到 APPLE 和 UCSD PASCAL 所提供的額外的非易傳的特性。

記錄比較 (Record Comparisons) APPLE 和 UCSD PASCAL 可以比較記錄變數 (record variable) 相等或不相等；標準 PASCAL 則要求記錄的比較，必須每一欄都比較之，例如：

```
<type>  
  cardrec = record  
    suit: (SPADE, HEART, DIAMOND, CLUB);  
    rank: (ACE, KING, QUEEN, JACK, TEN, NINE, EIGHT,  
          SEVEN, SIX, FIVE, FOUR, THREE, TWO)  
  end;  
  
<var>  
  card1, card2: cardrec;
```

在 APPLE/UCSD PASCAL 中，可以下法直接比較：


```
...  
if card1 = card2 then  
...
```

在標準PASCAL中，必須改寫如下：

```
...  
if (card1.suit = card2.suit) and (card1.rank = card2.rank) then  
...
```

十的冪 (Powers of Ten) APPLE和UCSD PASCAL 提供一個內建函數 **pwroften**，該函數回轉 (非負數) 的十的冪，**pwroften** 函數常式如下：

```
function pwroften(n: integer): real;  
( Return 10 to the nth power )  
  
begin ( pwroften )  
  if n <= 0 then  
    pwroften := 1.0  
  else if odd(n) then  
    pwroften := 10.0 * sqr(pwroften(n div 2))  
  else  
    pwroften := sqr(pwroften(n div 2))  
end;
```

本版的 **pwroften**，雖然可能需要修飾，以核對 **n** 的值是否溢位，但它並未這麼做，這個留給讀者做練習。

隨機數目 (Random Numbers) APPLE PASCAL 在系統館的 **applestuff** 單元內，提供一個隨機數目產生器，它在第二章內的 **theseus** 程式中被使用。而隨機函數 **random** 回轉一介於 0 到 32767 之間的整數。

被用來產生隨機數的方法有很多 (詳見推薦讀物)，下列 PASCAL 常式與 APPLE 提供的常式有相同的演算法：

```
function random: integer;  
( Return random integer in the range 0..32767 )  
  
var  
  i, j, carry, t: integer;  
  
begin ( random )
```

• 高等 Pascal 程式設計技巧 •

```
for i := 1 to 7 do begin
  carry := 0;
  for j := 1 to 4 do begin
    t := 2 * seed[j] + carry;
    carry := t div 256;
    seed[j] := t mod 256;
  end;
  if (seed[1] div 128) <> (seed[4] div 128) then
    seed[1] := (seed[1] + 1) mod 256;
end;
random := 256 * (seed[1] div 2) + seed[3];
end;
```

seed 是一個 4 位元組的陣列：

```
<var>
  seed: array [1..4] of 0..255;
```

本陣列爲了要保持其值在呼叫 **random** 時，無所損害，它必須是全盤的（**global**），在呼叫 **random** 之前，它必先設定初值，例如：

```
...
seed[1] := 90;
seed[2] := 178;
seed[3] := 246;
seed[4] := 147;
...
```

如果用上法把 **seed** 陣列設定初值，則我們每次都會得到相同順序之隨機數（許多環境下都希望如此）。爲了避免它，APPLE PASCAL 提供 **randomize** 程序，它會產生不同的隨機順序。此與呼叫 **random** 之前，改變 **seed** 陣列的元素，有異曲同工之妙；如何做，則隨你之意。例如，你可以衡量機器週（**machine cycle**）的兩鍵時間區間。

其他問題 (Remaining Problems)

其他的問題都非常困難，並不能藉設計易傳性來代表APPLE PASCAL 的內建資料型態，或是寫易傳性常式，來取代APPLE PASCAL 的內建函數，來解決問題。

輸入／出函數 (I/O Functions) APPLE 和 UCSD PASCAL 和真實世界的大多數之PASCAL 版本一樣，允許程式在作業系統之下接達下列輸入／出函數：

- 由名稱接達檔案。
- 鍵盤輸入沒有回音。
- 輸入／出錯誤偵測。
- 每次讀／寫「一頁」檔案。
- 將本文輸送到系統的印字機（不必宣告成檔案）。

APPLE PASCAL（但非UCSD PASCAL）也提供了 **keypress** 函數，該函數可以發現使用者打入的字是否已由程式讀進。

因為標準PASCAL無法完成這類函數，所以不能提供關於字串和長整數方面的易傳性的解答。我們曾嘗試在許多「基本的」常式 (**getkey**, **tfopen**, **lprint** 等等) 去將這些代碼獨立；把這類程式轉換成其他版本的PASCAL，多多少少要改變一些基本的常式。幸運地，因為有許多版本的PASCAL並無與此相當的服務常式，本工作經常只包括自APPLE的方言做翻譯。

如果你的PASCAL並未直接提供這些特性，可以直接在作業系統直接呼叫而得到這些特性。這可以透過組合語言副常式，或一（非易傳性的）內建**CALL**程序，使機械語言常式由位址呼叫之。例如，如果在CP/M作業系統下工作，這些

功能悉由呼叫 CP/M 的 BDOS 實行之。

關於上述有三種可能方式。許多版本的 PASCAL 提供與 **moveleft** , **moveright** 和 **fillchar** 等效的內建常式，而它們的名字又相同。若你的 PASCAL 有一適當的組合語言介面，而該常式非常簡單，又足以用組合語言來設計之。最後，一些不同的自由型態聯合技巧（在第八章的 **peek** 和 **poke** 內討論）可以經常在資料的傳輸時用於型態的核對。

記憶體的管理 (Memory Management)

在設計動態的記憶體的配置，和第八章的 **pascalc** 程式中的解除配置常式，廣泛的被 APPLE PASCAL 作業系統和內建常式 **mark** , **release** 和 **memavail** 等充份利用。但是在標準 PASCAL 就無法具有這些易傳性。

通常的 PASCAL 總是提供一些常式，以避免自己設計自己的一般性目的之記憶管理常式。第八章內曾提過，在 UCSD 的 IV.0 版本和它之後的版本，提供 **varnew** 和 **vardispose** 常式。

有 PASCAL 多少提供一個與 APPLE PASCAL 相似的記憶管理計劃：一堆疊 (**stack**) 和 **heap** 結構和 **mark** , **release** 和 **memavail** 程序。這些常式可能於第八章內用到，有的曾做些微的修改、有的全然不改。

控制使用 PASCAL 程式所使用到記憶體的量，總是由你的局部作業系統和由 PASCAL 所實行的運轉時間 (**run-time**) 常式聯合處理的。通常，爲了要瞭解你的「自由記憶體表列」，你必須建立、核對兩個系統的軟體文件。

特殊的語言特性 (Special Language Features)

除了非標準資料型態和常式之外，APPLE PASCAL 系統還提供下列額外的特性，使得設計師的工作更爲簡單：

- 分隔地編譯單元。 (**Separately compiled units**)

• 附錄 A 易傳性問題的探討與對策 •

• 段 (segment) 程序。

• 組合語言介面。

• 編譯器的選擇。

許多 PASCAL 的實施，如果與本書的這些特點不同，則必然提供另外一種相似的形式。例如，不使用分隔地編譯單元，每次編譯程式時消耗大量的原始代碼檔案，即降低模組性，增加編譯時間。如果避免用段程序，則記憶體的使用效率降低。你可以全部用 PASCAL 寫程式，而完全不使用組合語言，則你的代價是使程式變慢。

如果選用非易傳性的編譯 ({ \$ r - } , { \$ s + } 等等) 情況如何呢？一般來說，APPLE PASCAL 透過編譯選擇來達到這些功能，如你的 PASCAL 並未提供等效任選，則其提供的代碼要重寫。

硬體 (Hard ware)

本書最不具易傳性的觀念，是在靶標 (target) 電子計算機的硬體能力。所有的程式都是假設在一 CRT 為基礎的系統和交作的使用鍵盤輸入。許多程式假設是像磁碟一樣有大量的儲存體。如果不使用這種「最低的共同標準」硬體來翻譯這些程式，將是困難重重，其結果是可能不具滿足感。

我們用到 APPLE II 的圖形和音響能力。如你的電子計算機有相當的硬體，而你的 PASCAL 可以接達它們，則這些與硬體有關的常式很容易翻譯在你的系統上使用。

推薦閱讀 (Recommended Reading)

在 PASCAL 中，定義一字串般的資料型態的另一種易傳性的方法，請看 Software Tools in Pascal (由 B.Kernighan 和 P.J.Plauger 合著)。

• 高等 Pascal 程式設計技巧 •

附錄內的大部份常式，已翻譯成組合語言。如你使用一 6502 或 Z-80 的微處理系統，請參考 6502 Assembly Language Subroutines 和 Z-80 Assembly Language Subroutine 兩書，這兩本書是由 L. Leventhal 和 W. Saville 合著的，都討論字串處理和 BCD 算術常式。

fixed 算術常式是依據 D. Knuth 所著的 **Sem numerical Algorithm** 而得到啓示的。這古典的工作包括隨機產生器和算術兩方面的資訊。